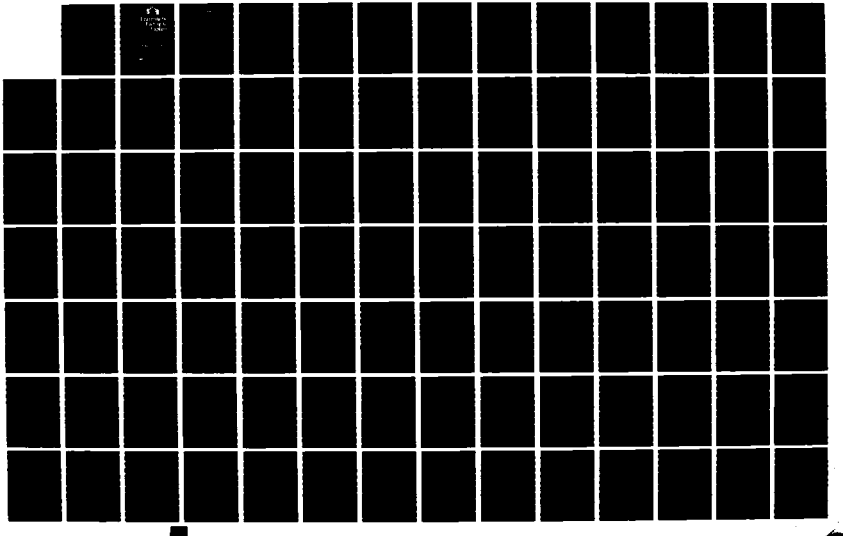
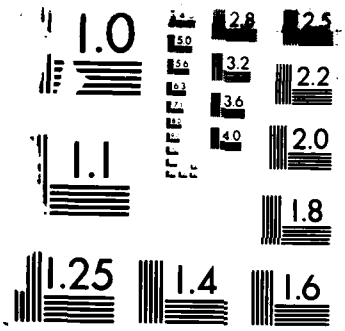
AD-A174 730 LEARNING BY FAILING TO EXPLAIN(U) MASSACHUSETTS INST OF 1/2
TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB R J HALL
29 MAY 86 AI-TR-906 N00014-85-K-0124

UNCLASSIFIED F/G 5/10 NL





MICROCOPY RESOLUTION TEST CHART

Technical Report 906

Learning by Failing to Explain

Robert Joseph Hall

MIT Artificial Intelligence Laboratory

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1 REPORT NUMBER AI-TR-906	2. GOVT ACCESSION NO. AD-A174 730	3. RECIPIENT'S CATALOG NUMBER
4 TITLE (and Subtitle) Learning by Failing to Explain		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert Joseph Hall		8. CONTRACT OR GRANT NUMBER(s) NSF Graduate Fellowship N00014-85-K-0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE May 29, 1986 13. NUMBER OF PAGES 140
14 MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Learning, Design, Explanation, Graph Grammar, Heuristic Parsing, Subgraph Isomorphism.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Explanation-based Generalization requires that the learner obtain an explanation of why a precedent exemplifies a concept. It is, therefore, useless if the system fails to find this explanation. However, it is not necessary to give up and resort to purely empirical generalization methods. In fact, the system may already know almost everything it needs to explain the precedent. Learning by Failing to Explain is a method which is able to exploit current knowledge to prune complex precedents, isolating the mysterious

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (cont.)

parts of the precedent. The idea has two parts: the notion of partially analyzing a precedent to get rid of the parts which are already explainable, and the notion of re-analyzing old rules in terms of new ones, so that more general rules are obtained. Keywords:

FLD F1

Learning by Failing to Explain

by

Robert Joseph Hall

This report is a revised version of *On Using Analogy to Learn Design Grammar Rules*, a thesis submitted to the Department of Electrical Engineering and Computer Science in December 1985, in partial fulfillment of the degree of Master of Science.

Learning by Failing to Explain

by
Robert Joseph Hall

ABSTRACT:

Explanation-based Generalization requires that the learner obtain an explanation of why a precedent exemplifies a concept. It is, therefore, useless if the system fails to find this explanation. However, it is not necessary to give up and resort to purely empirical generalization methods. In fact, the system may already know almost everything it needs to explain the precedent. *Learning by Failing to Explain* is a method which is able to exploit current knowledge to prune complex precedents, isolating the mysterious parts of the precedent. The idea has two parts: the notion of partially analyzing a precedent to get rid of the parts which are already explainable, and the notion of re-analyzing old rules in terms of new ones, so that more general rules are obtained.

This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

Acknowledgements

I would like to thank all of the people who have helped me with discussion, comments, criticism, and encouragement about this work: Patrick Winston, my advisor, for insight and encouragement; Peter Andreae, Dan Brotsky, Rich Doyle, David Kirsh, Rick Lathrop, Jintae Lee, and Brian Williams for technical discussions and hearing the whole thing many times; Tomas Lozano-Perez and Michael Brady for insightful criticism; and the AI Lab in general for providing a good place to do this type of work.

Of course, I sincerely thank my loving wife, I-Fan, for moral support, encouragement, and gluing most of the figures; my family, for everything from support and encouragement to postal chess, weekend phone calls, and Cal football reports. Thanks also to the Osmium Woodwind Quintet, a rare group of friends.

Contents

Chapter 1

Introduction and Overview

A primary motivation for learning from precedents is the intuition that it is easier for a domain expert to present a set of illustrative examples than it would be to come up with a useful set of rules. Explanation-Based Learning Methods use explanations of why a precedent exemplifies a concept in order to find a weaker precondition for which the explanation of concept membership still holds. This weaker precondition describes the generalization of the precedent. Mahadevan[9] has applied this to logic design. Smith, *et al*[17], have applied explanation-based techniques to knowledge base refinement. Mooney and DeJong have applied it to learning schemata for natural language processing[13]. Winston[21] abstracts analogy-based explanations to form rules.

This notion of using a domain theory to guide generalization is a powerful way of finding justifiable generalizations of concepts, in contrast to the unjustified leaps made by purely empirical methods. Unfortunately, the problem of explaining things is hard; after all, theorem proving is a special case of it. Thus, any method which tries to learn in complex domains is bound to fail at explanation a good part of the time. There are at least two reasons why an explainer can fail: the theory is incomplete, so that there is no explanation; or the explainer simply can't find the explanation, even though it exists. The latter case is not just mathematical nitpicking: the complexity of VLSI circuits and the rich set of optimizations possible creates large problems for any circuit-understander.

On the other hand, it is seldom the case that a learner knows absolutely *nothing* about an example it fails to explain; frequently, a small mysterious thing comes embedded in a large, mostly well-understood example. For instance, consider a multiplier circuit where the only difference between its design and a known one is in the way one particular XOR gate is implemented. It would be a shame to retain the complexity of the entire multiplier when the only new structural information was in one small subdevice. Rather than just reverting to completely empirical techniques when the explainer fails, the learner needs some method for focusing attention on the new information contained in the precedent. That is, the efficient student should, as much as possible, *know what it is that he doesn't know*. The student who is able to say, "I don't understand step 5" learns more quickly (and less painfully) than the student who is only able to respond, "Huh?" This notion of

pruning away parts of the precedent which are explainable is what I call *Learning by Failing to Explain*[1]. It is a complementary notion to explanation-based learning: the former operates precisely when the latter fails, and when the latter succeeds there is no reason to try the former.

There are at least two techniques which comprise Learning by Failing to Explain: the first is where the learner analyzes the given precedent as much as possible, then extracts the mysterious part as a new rule (or pair of rules). I call this *Precedent Analysis*. The second technique uses new rules to re-analyze old rules. That is, Precedent Analysis needn't be applied only to precedents; there are cases where it is beneficial to have another look at (possibly over-specific) rules found previously. This is called *Rule Re-analysis*.

Methodological Note

This work is not a study of human learning or human designing. While being motivated in some cases by intuitions about human behavior, it does not claim to model it in any way.

1.1 Overview

The main goal of this work is to study how current knowledge may be used to constrain the generalization of examples. A second goal of this work is to explore the representation and use of Design knowledge. This is both because design is interesting in itself, and because it is important to know that a learning system is acquiring useful knowledge. Thus, the domain of the current system's learning is design knowledge. In particular, the system learns structure/function knowledge, that is, knowledge about which structures implement which functions. It is agreed at the outset that this is not all the knowledge necessary to succeed at design; however, it is argued below that this type of knowledge can be usefully applied in concert with other types (for example, search control knowledge and analytic knowledge) to produce interesting design behavior.

The system uses a *Design Grammar* to represent structure/function knowledge. As a formal system, a Design Grammar is defined similarly to a string grammar, with the difference being that the "elements of the language" are not strings (*i.e.* linear graphs) but arbitrary graphs which represent functional blocks and interconnection. The learning system learns Design Grammar rules from precedents.

A Design Grammar is an interesting representation of structural design knowledge both because it is learnable from examples via the methods described here[2], and because it enables four interesting design competences:

- *Top-Down Design*: the ability to take a relatively high level specification of the function of a device and refine it successively by choosing implementations of subfunctions, then refining the refinement, and so on.

[1] Note that "failing to explain" connotes something stronger than "not explaining" —and presumably other methods as well

- *Optimization*: the ability to take one device and replace a piece of it with some other piece so that the resulting device is functionally the same.

- *Analysis*: the problem of establishing a justification for why some device performs some given function.

- *Analogical Design*: the ability to solve a new problem in a way similar to some already solved problem, or by combining elements of the solutions to many old problems.

The current system has been run on examples from two actual design domains. The two experimental domains I've chosen to study are CMOS world and Gear World. The former is a simplified version of a digital logic circuit domain in which the basic building blocks of devices are CMOS transistors. (Issues such as speed, area, and cost are not taken into account explicitly.) There are two types of transistors: the PTRANS takes three inputs and acts like a negative-active switch (if the "gate" input is 0, then the "source" and "drain" outputs are connected by a wire. Otherwise they are unconnected). The NTRANS also takes three inputs but is active when its gate is connected to 1. These combine with power and ground connections to yield Boolean functions.

The Gear World is a simple version of the realm of gears, sprockets, chains and shafts. Gears are taken to be circles with infinitesimal teeth that always fit together. Sprockets likewise are circles with infinitesimal teeth that always fit into the chain. These may be mounted on shafts, possibly more than one to a shaft. They combine functionally to produce angular speed ratios between input shafts and output shafts.

The essence of the learning mechanism to be presented here, Learning by Failing to Explain, is that the learner should not just accept something it can't understand as a new thing to remember; rather, it should try to understand as much of the mysterious thing as possible and then formulate a conjecture about what is new.

Learning from an example design is a matter of deciding what previously unknown techniques were used by the designer. The approach taken here is to partially reconstruct the problem solving process used by the designer by recognizing instances of known techniques, then conjecturing that the difference between the partially reconstructed solution and the entire solution can be explained by a single transformation. As will become apparent later, this last conjecture step is based solely on syntactic similarity and can lead to false conjectures.

In Design Grammar terms, the process of reconstructing the design process is the problem of *parsing* the design. Thus, the task of the system is to parse the example "as much as possible." This leads to the notion of a *maximal partial parse* and a heuristic algorithm for finding one.

Examples may actually be constructed using more than one technique which is unknown to the learner. Thus, the maximal partial parse produced by the learner will not go as far toward understanding the precedent, leaving a conjectured rule which really consists of many techniques applied together. This rule is clearly not as general as the collection of single-technique rules would be. The system has no

way of inferring the more general rules. Later on, however, after analyzing more precedents, the system may be able to go farther in partially analyzing the old precedent, thereby obtaining a more general rule. On the other hand, the system has already analyzed some of the old precedent, so this needn't be redone. Thus, the rule derived from the old precedent is treated as a precedent and analyzed using the new rules. This process is known as Rule Re-analysis. I will show in a later Chapter that this process is more powerful, in that it leads to more general rules in fewer precedents, than just using Precedent Analysis alone.

1.2 Summary of the Main Ideas

In the pages to follow, I will

- define the notion of Design Grammar.
- show how the four competences arise from having knowledge encoded in a Design Grammar.
- explain Precedent Analysis, wherein the learner first applies its current knowledge to a precedent to deduce the essence of what is new about it, then makes a plausible conjecture as to a new design rule.
- explain Rule Re-analysis, wherein the system makes good use of examples by using new rules to re-analyze old rules.
- give a sufficient condition to ensure that the rule generated is true independent of context.
- explain the matching algorithm whose good performance is crucial to recognizing previously known substructures: *subgraph isomorphism via constraint propagation.*

Chapter 2

Scenarios

This Chapter presents scenarios illustrative of the two techniques which comprise Learning by Failing to Explain. Due to the forward reference problem, it may be advisable to skim this Chapter on first reading. It should be possible to get the general idea without understanding the details of the representation.

2.1 A Note on Diagram Interpretation

Many of the Figures in this document were produced by the system which implements Learning by Failing to Explain. In order that the reader can understand them, it is important to understand how the program produces them.

The system has only a rudimentary method of placing graph nodes and routing arcs between them. The scheme it uses is to pretend that the graphs are all forests, with data flowing upward from leaves to roots. (See Figure 2.1 for an example.) It therefore places all output connection points on the highest row (those numbered 6 and 7), with the functional blocks driving those on the next row down (those numbered 11 and 9 in the left graph and 12 in the right graph), the connection points input to those on the next row, etc. Forest edges are drawn between the nodes so placed. The direction of dataflow along an arc goes toward the end of the arc closest to the black dot[1]. Any non-forest graph will have edges which either flow downward or flow between children of different roots (in Figure 2.1, node 6 in the right graph is both an input and an output to the circuit, so it has a non-forest edge flowing down into 14). These are added as an after thought by the drawing program. There is no attempt to route arcs around obstacles; hence, there may occasionally be labels overwritten by other labels.

The labels in the boxes indicate what type of node the box represents. Any label prefixed by "FB-" represents a functional block of the type following the hyphen. Any label which is just a number is a connection point (always of type bit in this implementation). Arc labels appear midway between the endpoints of the arc. Arc labels are prefixed by their input output type. Any box with just a period in it should be ignored; consider it, together with the two arcs incident with it, to be

[1] There is a small black dot 90% of the way from one end of an arc to the other.

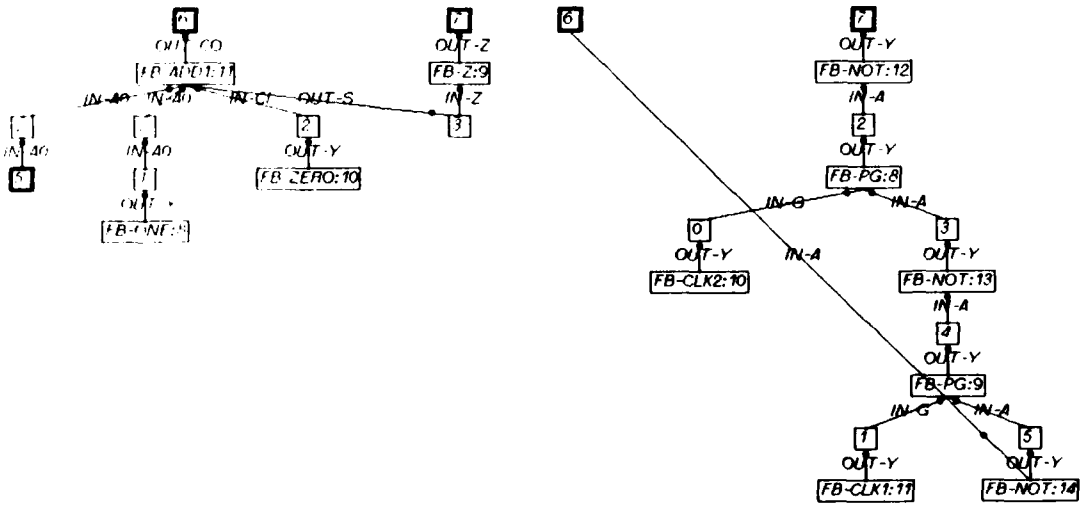


Figure 2.1: Precedent Analysis Scenario Precedent: the left graph is a high-level description of a one-bit incrementer with delayed output; the right graph is a low-level, optimized description.

one arc of the same type as the incident arcs (their types will always coincide). These nodes are put in for reasons having to do with unimportant details of the implementation.

Highlighted nodes (ones with thick boxes) indicate isomorphisms and correspondences between the two graphs. That is, the sets of highlighted nodes in each of the two graphs correspond in some way, usually in being isomorphic subgraphs. (In Figure 2.1, the highlighted nodes are the circuit inputs and outputs.)

Function Names. Most of the functional block names have their usual meanings from logic, *e.g.* NAND. MUX stands for the multiplexor block, which has three inputs and one output. If the select (s) input is 0, then the output is equal to the value on the $a0$ data input. If s is 1, then the output is equal to the value on the $b1$ input.

ADD1 stands for a one-bit addition cell with carry input and carry output. That is, it has three inputs (a, b, ci) and two outputs (s, co). The s output has the value $a + b + ci \bmod 2$, and the co output has the value 1 if at least two of the three inputs are 1, 0 otherwise.

ADD2 stands for a two-bit addition cell, likewise with carry in and carry out. This implements two-bit, twos-complement addition.

PG stands for a pass gate block. (This corresponds to a CMOS transistor.) It has two inputs and one output, where the values come from the set $\{0, 1, X\}$, X standing for high impedance. When the g input is 1, the d output is equal to the value on the s input. When the g input is 0, the d output has the value X.

Z stands for a single unit of time delay. (The text may sometimes refer to this as a Z^{-1} block.) Its single output is equal at time t to the value of its single input at time $t - 1$.

CLK1 and CLK2 are functional blocks representing clock generators. CLK1 represents a phase 1 clock signal, and CLK2 represents a phase 2 clock signal.

2.2 Precedent Analysis Scenario

Suppose the Learning by Failing to Explain system is given an initial knowledge base consisting of the (CMOS World) Design Grammar of Appendix D. This means it has, *a priori*, those rules available for analyzing precedents.

Next, suppose the system is shown the precedent in Figure 2.1. A precedent consists of two different descriptions of the same device, together with the variable correspondences. The variable correspondences are indicated by highlighting the nodes in the Figure. The Figure is somewhat ambiguous in that it is not explicitly indicated which node on the right corresponds to which node on the left. This was done to reduce clutter on the diagram. The system knows this information; it is simply not shown in the diagram. In some cases, the diagram may be augmented by hand to indicate the actual correspondences.

Figure 2.1 represents two different descriptions, the left one high level and the right one low level, of the same device. This device is a "one bit incrementer with

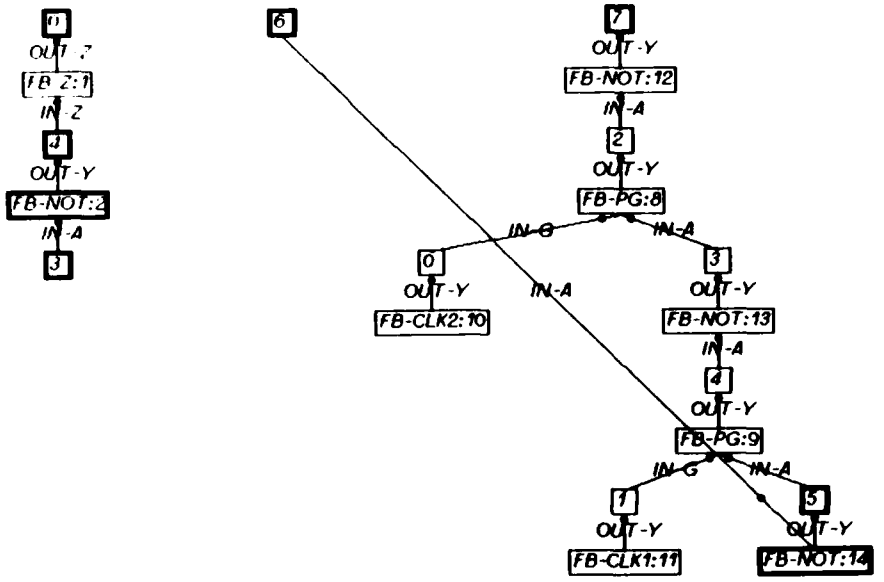


Figure 2.2: An Equivalence Derived from the Precedent Using Grammar Rules: the system has deduced, through grammar derivation, that the left graph is a statement of the function of the right graph. The highlighted subgraphs are deemed to perform the same subfunctions in the two graphs.

delayed output." Such a device might be found as a bit slice in an incrementer circuit, used for incrementing the program counter in a CPU.

The left hand graph is a high level functional description of the device: it indicates that the single bit input is to be added, using the one-bit add cell, to the constant ONE, with carry input ZERO. The result bit is then fed through a delay box (the FB-Z box).

It should be clear that the system can not fully explain, in terms of a grammar derivation using the rules in Appendix D, why the right hand graph implements the same functionality as the left hand graph. One reason is that there are no rules known to the system which involve an FB-Z box in any way.

On the other hand, it *can* derive that the left hand description in Figure 2.2 is functionally equivalent to the left hand graph in Figure 2.1. This involves a somewhat lengthy derivation of 25 steps. See Appendix E for the derivation and further examples of the implemented system's behavior.

Once the system has reached this point, the reasoning is as follows. Notice that Connection Point 4 in the left graph of Figure 2.2 is driven by the *syntactically identical* function of the corresponding inputs as Connection Point 5 in the right graph. Since the overall functions of the two graphs are the same, and these two connection points are constrained always to be equal, the subdevices corresponding

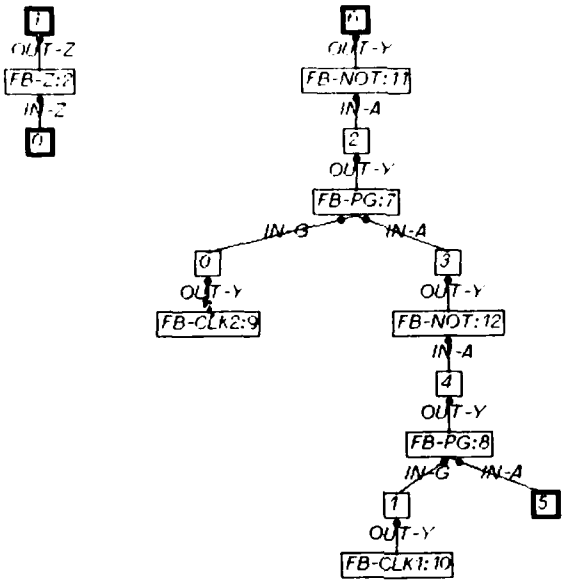


Figure 2.3: Rule Resulting from Precedent Analysis

to the complementary subgraphs of these devices must compute similar functions. The system therefore extracts those two subgraphs and induces a new grammar rule. The new rule is shown in Figure 2.3.

This is in fact a correct, new rule which represents an implementation rule for a delay function box.

2.3 Rule Re-analysis Scenario

Rule Re-analysis is a simple idea. This section illustrates it with a simple example. A later Chapter presents some of the subtleties involved.

Suppose, first of all, that the system is without any rules. Suppose, next, that the system is presented first with the precedent shown in Figure 2.4. (This represents two descriptions of the function AND(NOT($a1$), $a2$, $a3$).) Since the system has no rules, it can not analyze this precedent; that is, it can not apply any derivations to it in order to construct an explanation. It therefore simply stores the precedent as a new rule. The system now has one rule.

Suppose that the system next gets the precedent shown in Figure 2.5. (This is simply an implementation rule for NOR.) Clearly, neither side of the system's only rule (the one corresponding to the first precedent) is a subgraph of the second precedent. Therefore the system can not explain the second precedent, either.

This is where Rule Re-analysis enters the picture. If the system were now to

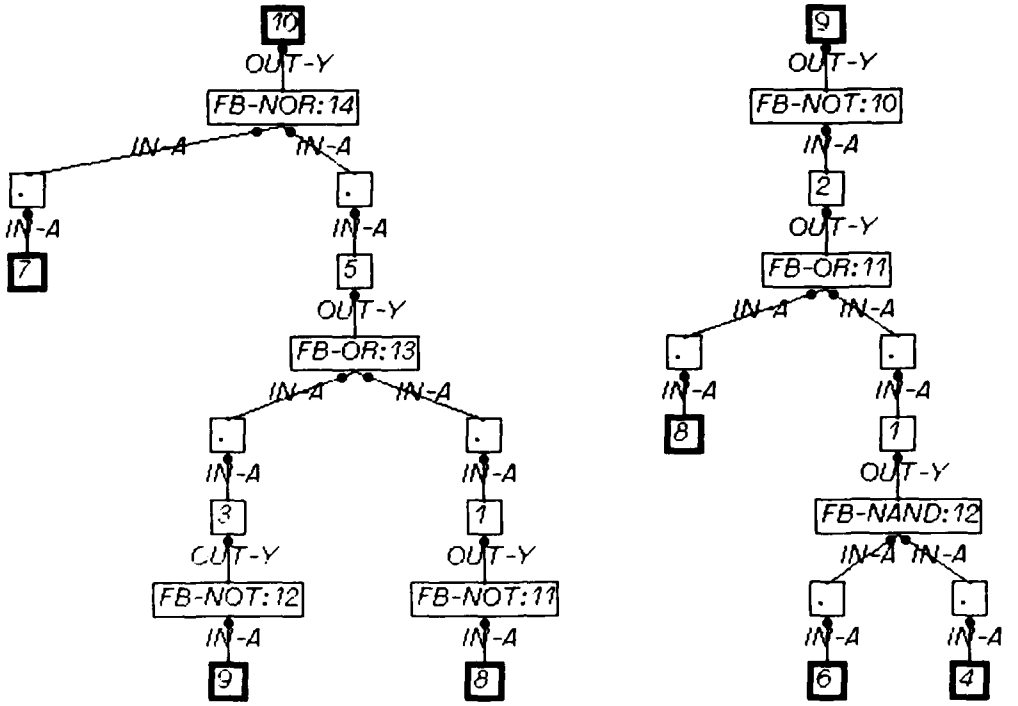


Figure 2.4: First Rule Re-analysis Precedent: the graphs are functionally equivalent to a three-input AND with one input complemented.

reconsider the *first* precedent, which is now a rule, it could succeed at Precedent Analysis. In fact, with a single application of the second rule, it could derive the equivalence shown in Figure 2.6. The partial match deduced is shown highlighted. Precedent Analysis would then extract the more general NAND rule shown in Figure 2.7. It should be obvious that the rule set so obtained, containing the NAND and NOR rules in addition to the rule corresponding to precedent 1, is more desirable than that without the NAND rule, because it allows the derivation of strictly more equivalences.

This method of using Precedent Analysis to re-analyze previously inferred rules is the essence of the technique of Rule Re-analysis.

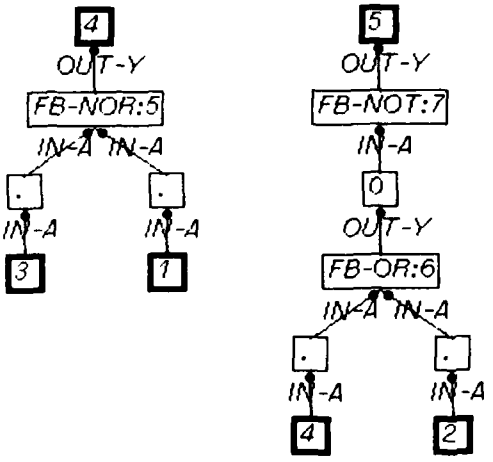


Figure 2.5: Second Rule Re-analysis Precedent: this is simply a NOR grammar rule, presented as a precedent.

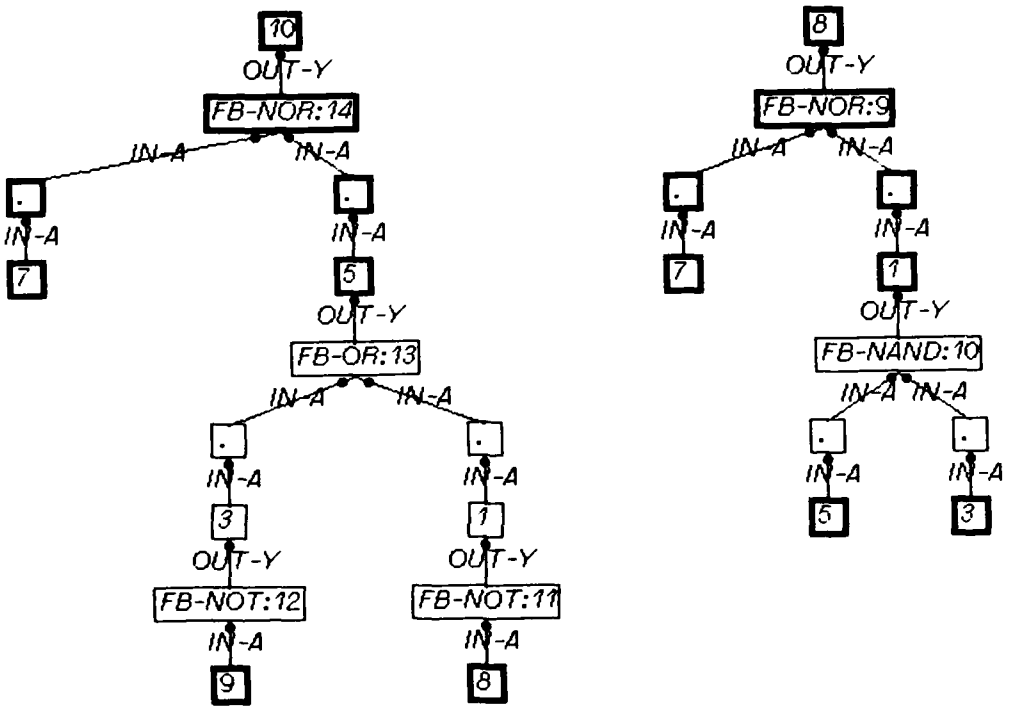


Figure 2.6: Intermediate Derived Equivalence: using Precedent Analysis, the system deduces this intermediate equivalence by looking back at the rule concluded from the first precedent (which was just that precedent) and applying the second precedent.

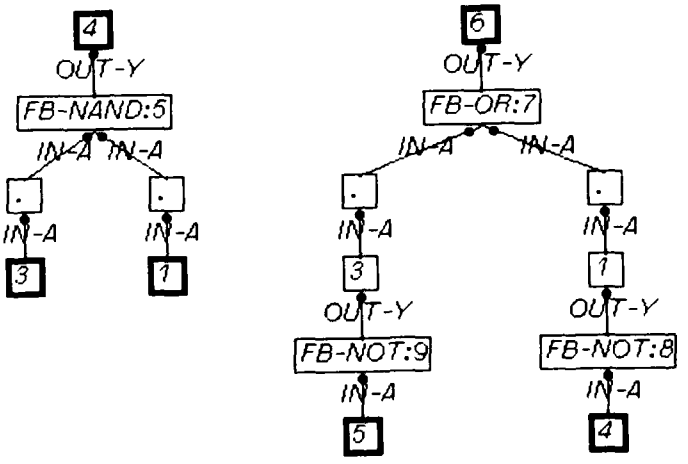


Figure 2.7: Deduced Rule, Using Rule Re-analysis

Chapter 3

Design Grammars

A Design Grammar is intended to encode knowledge about structure and function. Specifically, it records different implementations of given functions.

A Design Grammar consists of a set transformation *rules*, which associate one description of a device with another. It is somewhat reminiscent of a context-free string grammar in that it consists of rules whose left-hand side is a non-terminal symbol and whose right-hand side is something consisting of symbols that may be either terminals or non-terminals. There are two important differences: the right-hand sides of the rules in a design grammar are graph structures instead of strings, and the rules may be used in either direction in producing a terminal graph.

Appendix D has an example of a Design Grammar actually used by the prototype system.

3.1 Some Justification of This Approach

As in most work in the field of Artificial Intelligence and certainly all work in Machine Learning, the researcher must attempt to cleave a problem into a tractable chunk that is small enough to work on, yet that may still combine with other work to accomplish the entire goal. Thus, every AI researcher makes an assumption about how the larger problem may be decomposed and then picks one subgoal to work on. This work is no exception. It is the purpose of this section to make these assumptions explicit and to justify them.

3.1.1 Separating Out Structure and Function

Design is a vast AI problem. Consider all the factors that go into designing some device: say a gear train that transfers power in a high performance machine. The specification of the problem begins with stating the desired gear ratio between input and output. But in addition, the whole device must fit into some particular (possibly very strange) space. It must not break under vibrations of less than a given amplitude. It must stand up to temperature extremes. It must not cost too much, and so on.

In this thesis, I choose to separate out the subproblem of generating candidate implementations of functional specifications. (In the gear train example, this would correspond to generating a set of possible structures that give the correct transfer ratio.) The other subproblem, the one I don't deal with directly, is searching this space of candidate implementations. Epistemologically speaking, the idea is that knowledge about structure and function, *i.e. what can implement what*, can be separated from knowledge about controlling search through the space of implementations. The system encodes knowledge of structure and function in a grammar formalism, and learns the grammar rules from precedents.

3.1.2 A Precedent in the Literature

Ressler[15] has taken a similar approach in his thesis on op-amp design. He showed how knowledge about how to build operational amplifiers that meet certain specifications can be encoded in a grammar structure.

His program had other knowledge that helped constrain the search among the different types of amplifiers generated. There was a parameter analysis portion of the program that looked at a candidate design choice and tried to see if it could possibly satisfy the specs. If not, then yet another source of knowledge, his failure rules, took over to suggest where next to look for a solution.

This shows how knowledge of structure and function can be separated from knowledge about controlling search. His grammar generates a space of candidate designs that are indexed by certain specifications (my system uses only *functional* specifications, where his uses other considerations). The analytical and empirical (failure) knowledge is then used to guide the search.

3.1.3 How Other Knowledge Might Be Added

This work may be viewed as a generalization of the epistemological idea contained in Ressler's paper. The idea is to encode knowledge of structure and function in a grammar formalism. Then other knowledge, which can help guide the search, may be added as it becomes available.

Ressler's analytical component might be seen as a kind of "static evaluator," à la chess-playing programs. So one possible way of adding knowledge to the system would be to develop a lookahead-and-evaluate paradigm, where the system tries all possibilities out to a certain depth of search, then evaluates each resulting graph to see which could lead to successful designs. It throws out those that it can show are inferior.

Another method for adding knowledge would be to incorporate the knowledge required for dependency-directed backtracking[1]. An example of this type of knowl-

[1] *dependency directed backtracking* is a technical term that is meant to denote the part of the system where whenever a search path ends in failure, only those decisions are withdrawn that had some part in causing the failure. This is as opposed to automatically withdrawing the most recent decision made. This last technique is referred to as chronological backtracking.

edge would be a rule like "if the device isn't durable enough, try replacing the belt-drive with gear meshes." This knowledge corresponds to Ressler's failure rules.

Analogical methods might also be added, like noticing when a problem is similar to a solved problem and using syntactic similarity to justify trying certain paths over others. For example, one might reason that the current device needs to be just as durable as a certain precedent that used only gears (for that reason), so restrict the search to solutions that only use gears and not belt drives or pulleys.

3.2 What a Design Grammar Is

The purpose of this section is to make the definition more precise. The subsequent Section will deal with how to use a Design Grammar.

3.2.1 S vs F and T vs NT

A Design Grammar is a special type of *graph grammar*. See Appendix A for the relevant definitions. The goal of the learning component of the system is to construct a graph grammar that encodes structure and function knowledge in such a way that the system may use the grammar (possibly together with other knowledge) to generate efficient, functionally correct designs. Rather than get embroiled in the controversy about what is the real and true difference between structure and function, I avoid defining those terms. Instead I will appeal to the intuitive meaning of each: structure is roughly how to build something, and function is the set of interesting constraints that something enforces between its inputs and outputs.

I will, however, distinguish between two different senses of the word "function." (These are my definitions, which may or may not correspond with the reader's intuitive definitions.) A *behavior* is a mapping from a set of inputs to a single output value, defined on every combination of input values. This is distinguished from a *role*: a mapping of inputs to **sets of allowable outputs**. Note that a behavior can be identified with a particular kind of role, namely a role with all singleton images. Also, for example, a function about whose value on input a we don't care is really a role that sends a to the set of all possible values.

A role r_1 *satisfies* a role r_2 if and only if the value-set of r_1 for a given input set is a subset of the value-set of r_2 on the same inputs. If role r_2 has multiple outputs, r_1 must put out at least the outputs of r_2.

For example, an AND gate has a unique behavior that maps pairs of bits to their boolean product. It satisfies many roles, however. It can, for example, fill the role of carry circuitry in a two bit adder, if the adder's input space is restricted to the case when one of the inputs is the constant 01. This is because with such a restriction, the carry signal is high exactly when the other input number is 3, which is exactly when the AND of its bits is 1. Note that if the other input to the adder were 2 instead of 1, then AND would be insufficient to fill the role of the carry circuitry.

A *functional block* represents a role; it is the black box form of it. That is, a functional block is a box which takes in certain inputs and puts out values constrained by the role to lie in certain sets of allowable outputs.

Design problems are of the form, "using elements from set E, implement role f." The elements of the set E are things like transistors, wires, gears, or chains: directly usable, known operators. This is as opposed to the role f, which is more like *square root*(x), *multiply*(a, b), or *compile-Fortran-program* (P)[2]. Now, it may be that the set E contains something called a "multiplier" which in fact implements that role, but I nevertheless distinguish between the operator, which comes from the set E, and the role, which does not.

E may also contain directly implementable combination operations, like "weld together" or "connect with wire."

Note that in describing the function of something it may be impossible to describe it with one function name from the vocabulary; it might be necessary to compose functions. For example, the function $(2x + 4y)(3z - 6)$ is represented in just such a manner as the composition of multiplies, adds, and subtracts. It may be, however, that there is not necessarily any variable whose value represents the intermediate value $(2x + 4y)$, so representing the function requires another representational primitive, the *connection point (cp)*[3]. A cp can be thought of as a "variable" in that it represents a value (hence it has a type), except that there need be no physical (implementational) counterpart to it.

The role f stands for constraints between inputs and outputs. These inputs and outputs must be measurable somehow to be useful, thus they must have some structural manifestation. I will call a measurable (hence implemented) cp of an E-element, which is constrained by the role of devices that contain it, a *variable*.

I define a *terminal element* to be something from the set E. I define a *non-terminal element* to be either a functional block name like "plus", or a cp.

3.2.2 Base Representation

At the lowest level, structure/function knowledge is represented in a kind of *semantic net*. In particular, a functional or structural description is a graph consisting of typed nodes and typed links. Each terminal element has a representation as either a node or a link. A *terminal graph* is a graph that consists only of terminal elements. For a more formal definition of graphs, see Appendix A.

Any functional block or cp is represented by a non-terminal node of appropriate type. A functional block has arcs emanating from it. These arcs correspond to

[2]The role of a compiler is not a behavior, in general. Given a source program, we will accept any of many possible behaviorally equivalent results.

[3]The attentive reader may have noted that the notion of composition is unclear with regard to general roles. Define the composition of two single-input, single-output roles to be the unique role which maps any member of the domain of the first role to the union of the output sets of the second role, the union being taken over all elements of the output set of the first role on the given input. This definition reduces to the familiar one when both roles are behaviors. Extend this in the natural way to multiple-input, multiple-output roles.

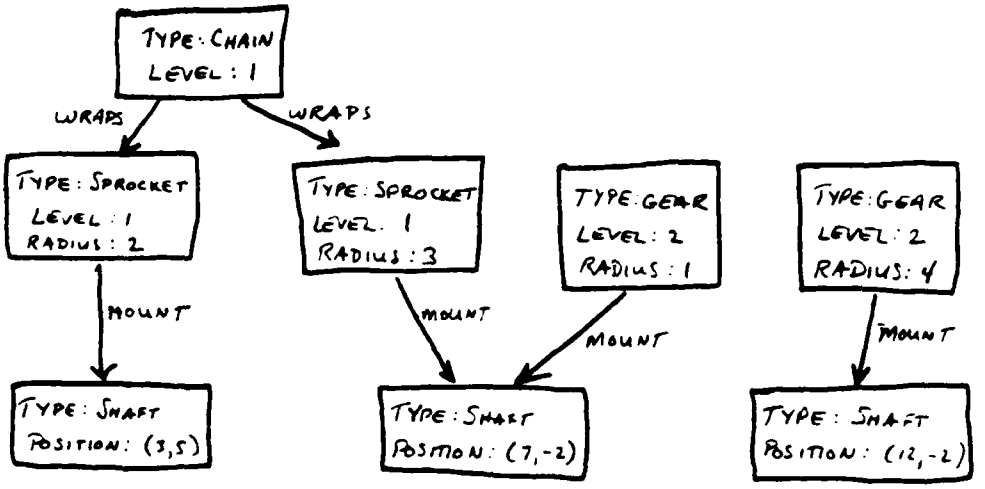


Figure 3.1: Terminal graph representation of gear train structure: this indicates that there are sprockets of radii 2 and 3 wrapped by a chain and mounted at level 1 on shafts at $(3,5)$ and $(7,-2)$. Also, there are two gears, one of radius 1 at level 2 on the second shaft, and one of radius 4 at level 2 on a shaft at $(12,-2)$.

"ports". Each type of port has a unique arctype associated with it. The arc points in the direction of data flow. Note that there may be more than one port of a given type.

Terminal graphs are intended to represent the lowest level of structural description of some device. Graphs consisting only of non-terminals represent the behavior of the device. There are hybrid graphs, and these have both structural and functional aspects. Typically, a device will have many levels of behavioral (or role) description, arranged hierarchically.

Note that it may be convenient to represent classes of structural or functional elements by a single node type with a parameter whose value distinguishes the members of the class. The class of gears, for example, is conveniently represented as a "gear" node with a "radius" parameter. Therefore it is possible to represent a class of devices with a single graph that contains parameterized elements and relations among the values. See Figures 3.1 and 3.2 for examples of graph representations of structure and function of a device.

3.2.3 Design Grammar Rules

As stated above, a Design Grammar consists of a set of rules. Each rule is of the form

$$LHS \rightarrow RHS$$

where LHS (Left-Hand Side) is a non-terminal graph (graph with exactly one non-terminal node) representing a behavior of the function denoted by the non-terminal.

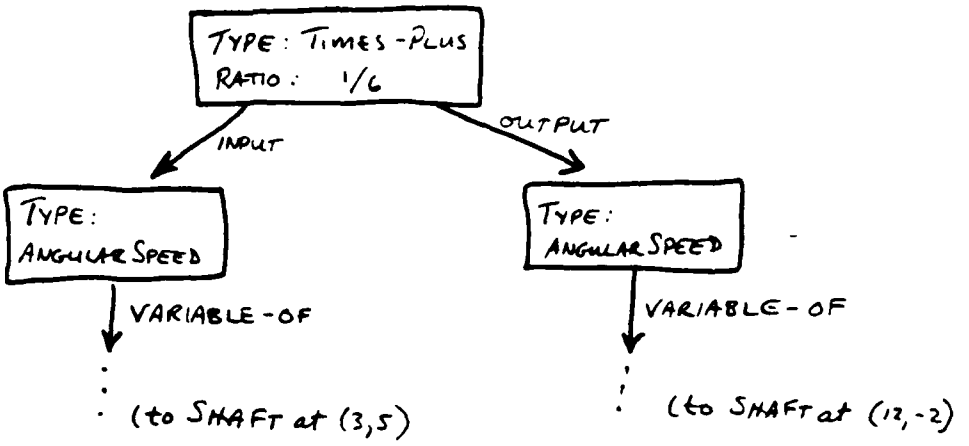


Figure 3.2: Graph representation of gear train function: this indicates that the angular speed of the shaft at $(12, -2)$ is one-sixth as large as that of the shaft at $(3, 5)$.

and RHS (Right-Hand Side) is a graph representing some *implementation* of the functional block. It represents that combination of the behaviors of its constituents compatible with the semantics of the connections. It therefore also represents an overall behavior.

Along with this comes an association between the input and output variables of the LHS and those of the RHS, which indicates which variables of the RHS correspond to which variables of the LHS. Thus, I am interpreting inputs and outputs as the connection points of the graph grammar rules. This needn't be a one-to-one correspondence[4].

It is not required *a priori* that the LHS represent the same behavior as that represented by the RHS. It is also useful to have rules around that associate things that only share some useful role, rather than an entire behavior. Thus, an implementation of "double-the-input" might be one that can only double integers between 0 and 15. These are not equivalent, because the implementation can't handle non-integers or too big or too small integers. But the role which is defined only on the range 0..15 might be a very useful one.

I will say that a grammar rule is *equivalence-preserving* if the LHS represents the same behavior is the RHS. See Figure 3.3 for an example of a grammar rule that does not preserve equivalence, and Figure 3.4 for an equivalence-preserving rule.

[4] *e.g.* one implementation of the non-terminal "buffer" (output = input) is simply a variable to which all connections are made. Thus both the input variable of the buffer and the output variable correspond to the same variable of the implementation.

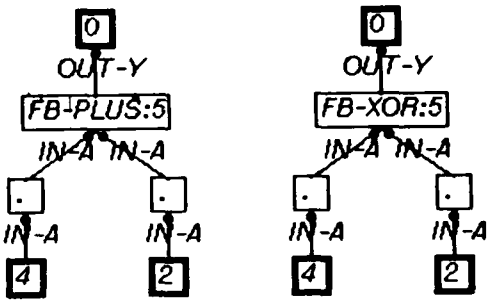


Figure 3.3: A non-equivalence-preserving grammar rule: it is only true when the values at the connection points are one-bit numbers (they might be arbitrary integers, for example).

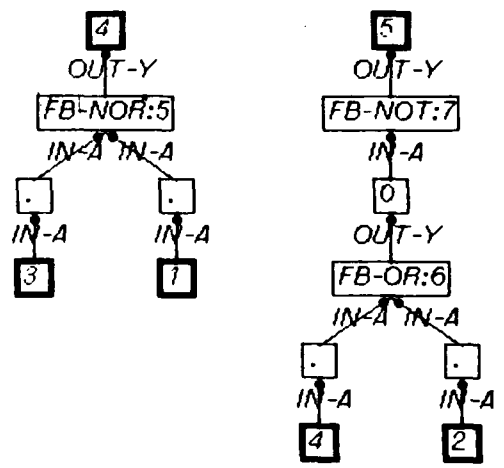


Figure 3.4: An equivalence-preserving grammar rule: this is true in all contexts.

3.2.4 Derived vs Primitive Rules

Consider the NAND rules in the Design Grammar of Appendix D. NAND has two equivalence-preserving implementations: NOT (AND (x,y)) and OR (NOT (x), NOT (y)). It would seem, at first glance, that these are two independent rules; however consider the following derivation.

NAND (x,y) $\equiv$ NOT (AND (x,y)

NAND, Version 1

$\equiv$ NOT (AND (BUFFER (x), BUFFER (y)))

BUFFER, Version 5

$\equiv$ NOT (AND (NOT (NOT (x)), NOT (NOT (y))))

BUFFER, Version 6

$\equiv$ NOT (NOR (NOT (x), NOT (y)))

NOR, Version 2

$\equiv$ NOT (NOT (OR (NOT (x), NOT (y))))

NOR, Version 1

$\equiv$ BUFFER (OR (NOT (x), NOT (y)))

BUFFER, Version 6

$\equiv$ OR (NOT (x), NOT (y))

BUFFER, Version 5

This shows that the Design Grammar gains no generational power by having both implementations of NAND: just as many graphs are generated having (either) one as are generated by having both[5].

There may be efficiency reasons for keeping around both implementations. However, it is also useful to record the sequence of rules that derives a rule. A *derived rule* is one that is not only derivable from others, but also has a known derivation. A *primitive rule* is any other rule. It is quite possible to have rules that are derivable, but recorded as primitive simply because no derivation is known.

A Design Grammar may contain both kinds of rule.

3.2.5 The Induced Role of a Subdevice

Suppose that some graph S, possibly a RHS or LHS of some rule, matches a subgraph of some graph, G. Then there is a distinguished role, $r_{S,G}$ which S implements in G. That is, G is composed of graph-elements that together represent a role f. For this to be true, S must map certain combinations of its inputs to certain acceptable sets of outputs. But this is precisely the definition of a role: a mapping of inputs to sets of acceptable outputs. If S has multiple outputs, then this is a mapping from input vectors to sets of output vectors.

Consider the set of all roles, h which satisfy the condition that if S is replaced by the block for h, then the resulting graph still implements f. For each input vector, form the union of the output sets of each of these roles. Then $r_{S,G}$ is the role which maps an input vector to this union.

[5] assume that the grammar already contains all the rules used in the derivation of ours

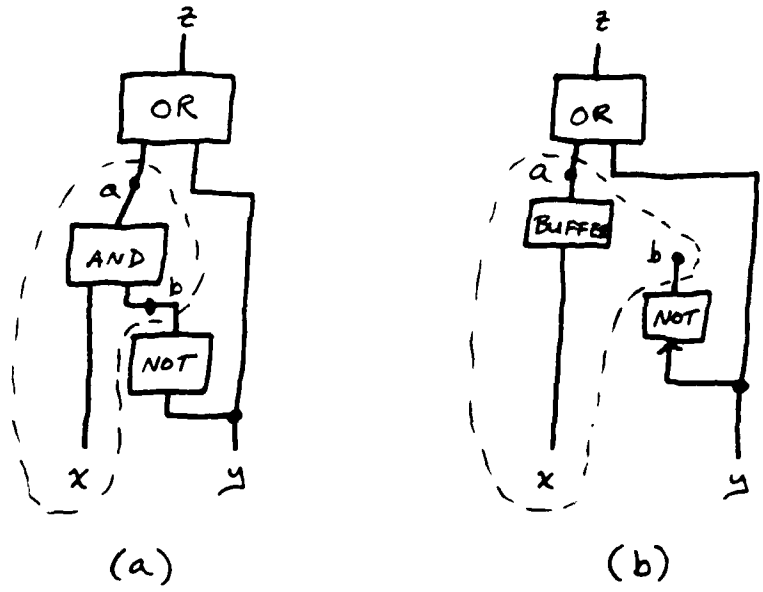


Figure 3.5: An Induced Role Example: the induced role of the subgraph enclosed in dashes is weaker than the AND box which fills it in the left graph; the overall circuit still works with the BUFFER, as shown in the right graph.

Call $r_{S,G}$ the *induced role of R in G*. This will become important later, as it helps shed light on when rules inferred by Learning by Failing to Explain represent allowable transformations.

As an example, consider the behavior represented in Figure 3.5 (a). Let G denote the overall behavior. Note that an equivalent statement of the behavior of G is that f = OR (x, y). Let S denote the subgraph enclosed in dashes; i.e. the one consisting of the AND box, a, b, and x.

Now, if y has value 1, then f is 1 regardless of the value of a. However, if y is 0, then f is equal to the value of a. But because y is 0, b must have value 1. Also, because G behaves like OR, a must be exactly equal to the value of x. Thus, $r_{S,G}$, a role defined on the inputs x and b and mapping to the sets of allowable values of a, must do the following:

$$r(x,b) \quad \begin{cases} \{0,1\} & b = 0, x = \{0,1\} \\ \{0\} & b = 1, x = 0 \\ \{1\} & b = 1, x = 1 \end{cases}$$

This is not identical to the behavior of the AND gate, which is used to implement the role. In fact, a BUFFER would satisfy this role equally well. See Figure 3.5 (b).

3.2.6 Allowability of a Rule Transformation

Suppose a rule is equivalence-preserving. And suppose that its RHS matches a subgraph of some other graph. Then the overall graph behaves precisely the same as the graph that has the LHS inserted in place of the RHS.

Suppose, more generally, that one side of a rule satisfies the role induced by the other side in the graph. Then, again, the overall behavior remains the same if the second side is replaced with the first side. When this condition obtains, I will say that the rule application is *allowable*.

The system can tremendously increase its power to generate implementations by using rules in all allowable directions. This leads, in Section 3.3, to a method of generating "optimizations" of devices.

The problem of determining when a rule application is allowable is a deep one; in fact, this thesis will not deal directly with it. I discuss this decision later in more detail.

A Design Grammar is defined to allow equivalence-preserving rules to be used backwards. In fact it is defined so that any *allowable* rule transformation may be applied. This is not really a change of the definition of graph grammar; it can be viewed as merely a bookkeeping convention, which avoids having roughly twice as many rules as necessary.

3.2.7 Generic Rules and Subtleties in Functional Representations

There is a class of domain-independent transformations which are always allowable. These arise out of the semantics of functions. The system should have these available in addition to the domain-dependent transformations. Because they are domain-independent, they can come "built-in". I term these *generic rules*.

The first such class involves the case where an instance of a side of a rule appears in the design, but two or more of its input connection points are the same connection point. (For example, PLUS (a, a).) This is a problem, because then the graph which represents the rule side (say PLUS (x, y)) is not isomorphic as a subgraph to the instance. The semantics of function, however, allow the substitution. Thus, special provision must be made in matching to allow for recognizing this case. When this occurs, it is necessary to alter the other side of the rule, usually, as there will be fewer inputs for it. This recognition and alteration process is relatively straight-forward; the difficulty is just in realizing that it must be done.

The second class of generic transformations involves output-sharing. Consider the behavior represented in Figure 3.6 (a). This is equivalent semantically to that represented in (b). The difference is that there only needs to be one AND node. I term the transformation represented by the move from (a) to (b) *output sharing*. The inverse transformation, from (b) to (a), I will call *node splitting*.

The currently implemented system handles these transformations implicitly. Whenever any rule is to be tried, if there are more connection points to the matched subgraph than exist in the rule's non-terminal graph, this means that there is po-

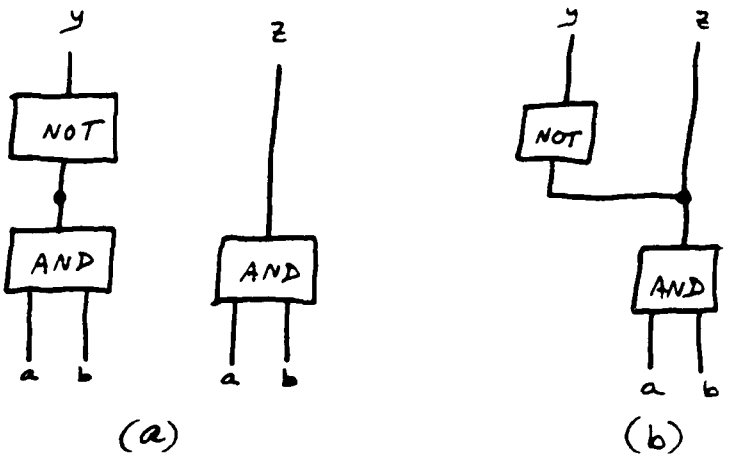


Figure 3.6: (a) AND Output Split, (b) AND Output Shared

tential for splitting off a subgraph. The system does just that. It allows the rule to be applied, but keeps all of the old nodes that contribute to (that is, are in the input graph of) some external node. It gets rid of all others.

For example, AND (1, x) matches the RHS of a BUFFER rule, so could be removed and substituted for. However, if the constant 1 is also tied to the input of another subgraph, the system performs the surgery as follows: remove the matched instance of the RHS, except leave the node representing the constant 1. Remove the connections from the remaining graph to the excised nodes (this removes one connection from the 1 node). Introduce an instance of the LHS graph into the remaining graph and merge the variables of the LHS-graph instance with the appropriate nodes in the remaining graph. Note that this has the effect of leaving the 1-node's other connection intact.

This would seem to create a problem if the rule's implementation contains feedback: in a feedback loop, all the nodes are necessary, as the input graph has a cycle. Tracing back from any output to see which nodes it depends on, one comes back to the original output: hence, all nodes in the cycle must be retained. Thus, no nodes are excised, so the same rule which was just applied, if its RHS consists of exactly the nodes of the cycle, is applicable again!

The smart system avoids this by noting that even if all the *nodes* must stay, there must be some *arc* that can be removed. This is the arc that drives the output variable of the rule. It can be replaced, because there is another arc from the new (replacement) non-terminal graph that will drive that node instead. The result will be equivalent functionally. Because the arc is missing, the rule will not be applicable in the same place again. *Voilà.* See Figure 3.7. In the Figure, once the arc is excised, some of the nodes disappear, because they are useless (they don't contribute to the value of some output of the system). These are represented as dashed boxes and arcs.

This introduces another class of generic transformations which are really just a bookkeeping convention that should be done whenever possible. Suppose in Figure

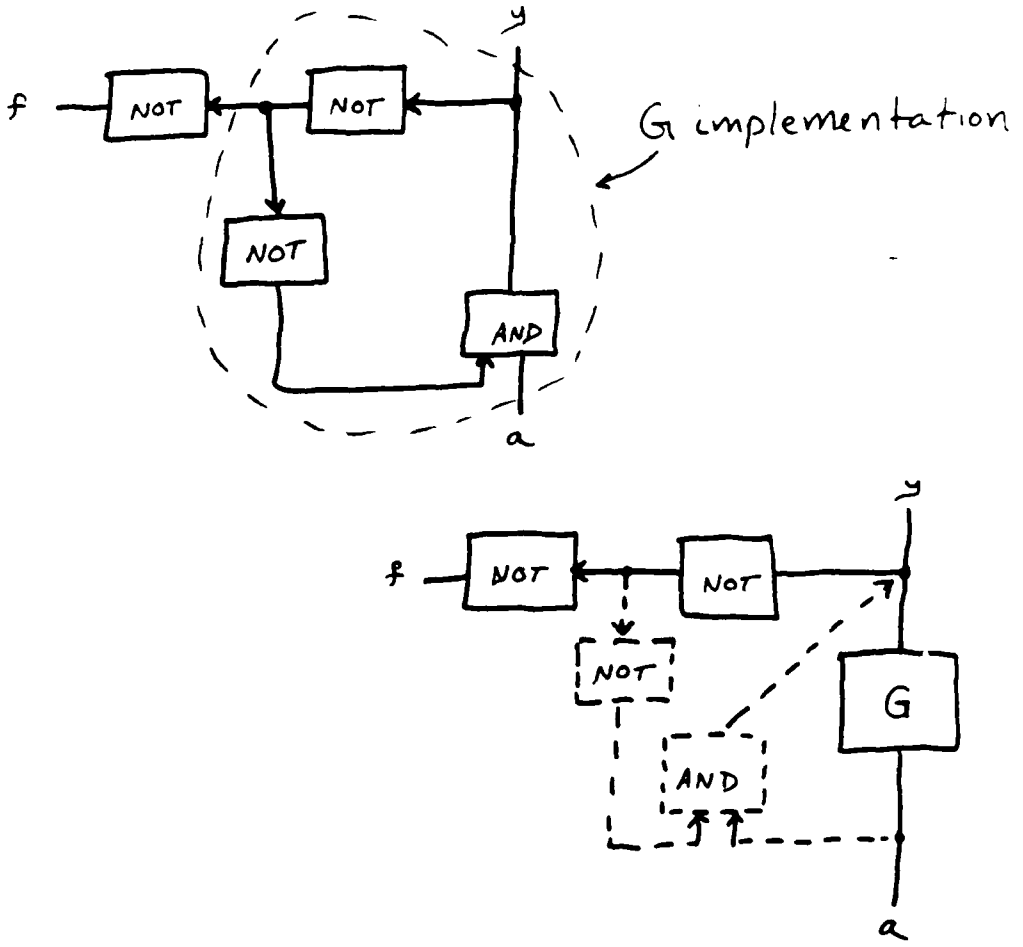


Figure 3.7: A Pathological Case of Automatic Node Splitting: as there is recursion in the input graph of f, none of the nodes would appear to be candidates for removal. However, the output *arc* of the AND is removable, because it is replaced by the output arc of the G box.

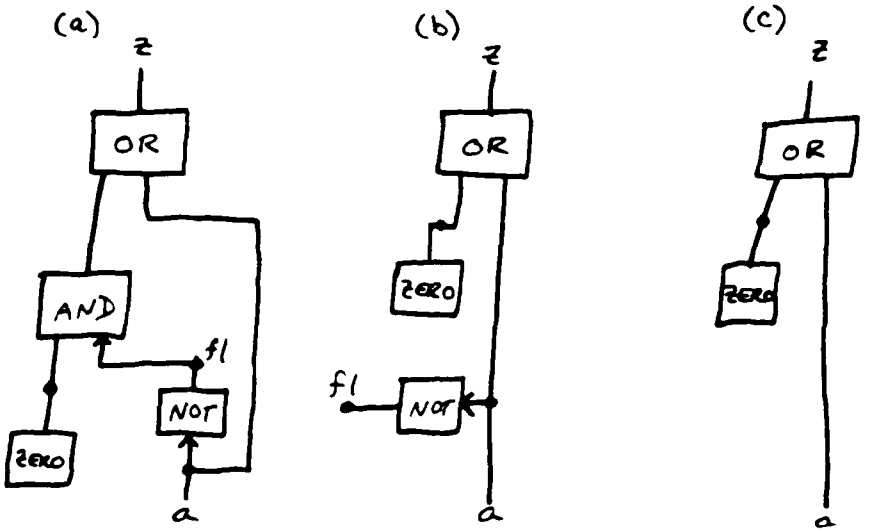


Figure 3.8: Excising Useless Pieces of the Graph: the NOT and $f1$ nodes in (b) are useless, because they affect no external or internal values.

3.8 (a) a transformation is made by replacing the AND (0, $f1$) subgraph with the ZERO non-terminal. That leaves the strange looking graph in (b). This is strange looking because $f1$ is a cp which has no physical (observable) counterpart. Therefore part of the graph is computing a value that is not used! This is semantically equivalent to the graph of part (c). If one allowed these loose ends to remain in graphs, there would be a proliferation of different representations of the same graph. This would be bad and useless. Therefore, the implemented system calls a simple routine for cleaning up these useless nodes after every rule application.

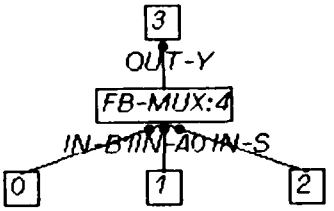


Figure 3.9: A Sample Design Problem: this indicates the desire for some implementation of the one-bit multiplexor function.

3.3 Why Want a Design Grammar

This Section will show how the four competences arise from having knowledge encoded in a Design Grammar. Examples throughout will refer to one of the particular design grammars presented in Appendix D. These are for the CMOS World and the Gear World domains.

3.3.1 Top-Down Design

Top-Down design arises quite naturally from a Design Grammar. The initial design problem is stated in the form of a graph representing the desired behavior. An acceptable answer is some terminal graph whose behavior satisfies the behavior of the original graph. In Figure 3.9, the desired function is that of a *one-bit multiplexor* circuit: it takes in a select bit s, data bit a, and data bit b. It puts out y, where y equals a if s is 0, and b if s is 1.

Using the *name* of the functional block as an index, *i.e.* "MUX", the system quickly retrieves all rules whose LHS contains the MUX functional block. The system then has a possible avenue to follow for each rule so retrieved. Choosing version 2, the task is done immediately: version 2 is a terminal graph implementation of a MUX. See Figure 3.10. For reasons other than functional, version 2 can be undesirable. So the system could then try version 1, which expands the MUX into a network of other blocks, see Figure 3.11.

At this point, instead of being done, the system must expand some more. Now, all rules are retrieved that have LHS containing any one of the non-terminals in the current (expanded) description. As it turns out, all blocks have exactly one implementation, so they are all used. These are all terminal graphs, so the resulting graph is the other possible implementation of MUX generated by the system. See Figure 3.12.

In the Gear World, the problem is essentially the same. However, because the elements have parameters and the rules have relations among parameters, an additional step is required. After choosing a candidate implementation, some parameter

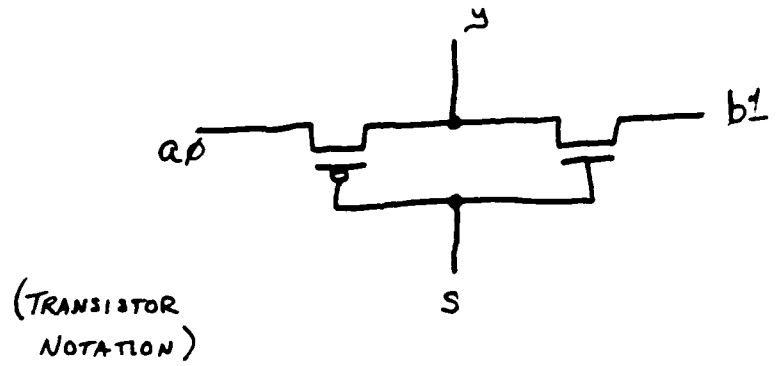


Figure 3.10: The MUX, Version 2: this is in CMOS transistor notation. This is a terminal graph.

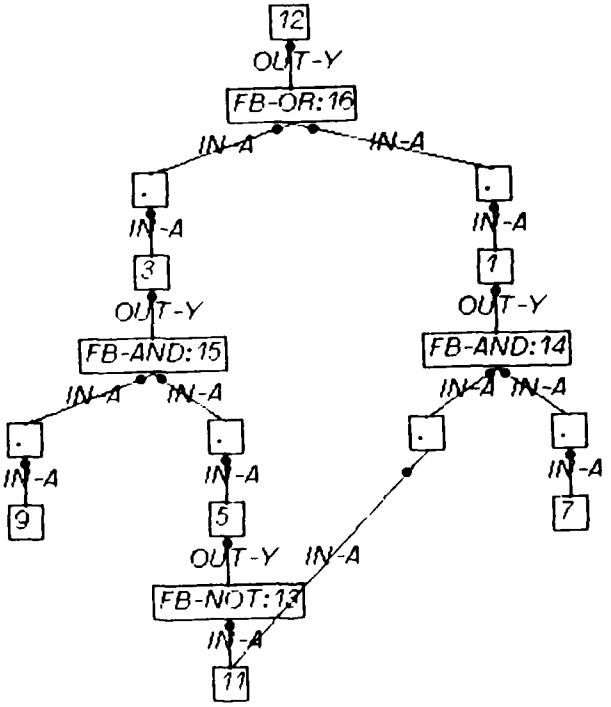


Figure 3.11: The MUX, Version 1: this is an implementation in terms of lower level functional blocks.

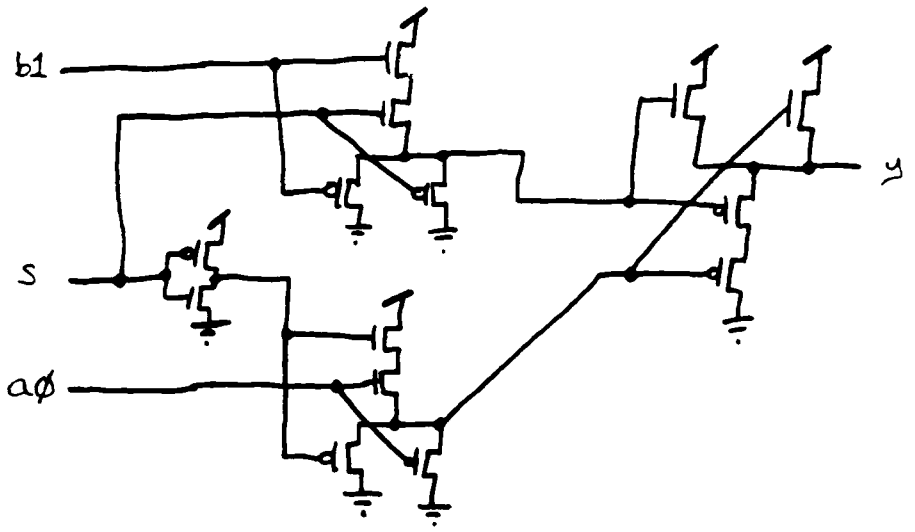


Figure 3.12: The Terminal Graph Resulting From MUX, Version 1: this is obtained by simply replacing each lower-level block with its terminal graph implementation.

analysis may be required before the next choice is made. This type of knowledge is not dealt with here. See Ressler[15] for an example of the use of this type of knowledge.

Of course, the simplicity of the example avoids the search control problem: whenever two or more implementations exist for a block, it has the potential to cause branching in the search tree. The control of this search requires additional knowledge, as discussed in Chapter 1.

Another problem exists with this scheme: the set of candidate designs generated with this method may not contain the best designs. Suppose in the previous example that input *a* were tied to the constant 1. Then the grammar would generate the same two terminal graphs, except that a connection to power would replace the input *a*. In the design resulting from version 1 of the MUX, much of the circuitry is unnecessary. In fact it is equivalent to the circuit in Figure 3.13, but the given design grammar can't generate this circuit from the MUX using only the top-down technique. The next subsection presents a solution to this problem.

3.3.2 Optimization

As mentioned in Chapter 3.2, if a rule preserves equivalence, then it makes sense to consider replacing either side of it with the other. And more generally, any allowable rule transformation (not just LHS replaced by RHS) makes sense in that it preserves

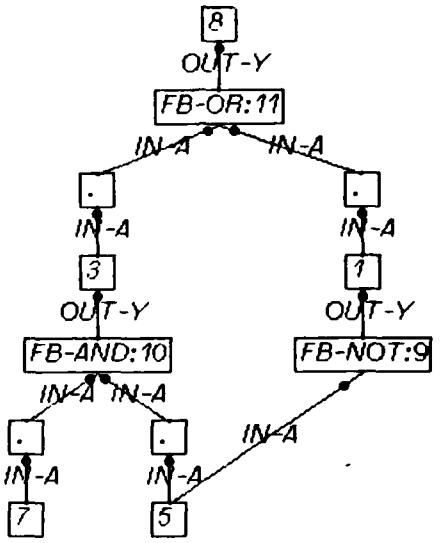


Figure 3.13: Optimized Version of the Modified Design Problem: this is an implementation of the MUX with one input tied to one.

the overall behavior[6]. By induction, any sequence of such transformations must preserve the overall behavior.

This enables optimizing a design: generate a first pass at the design by top-down techniques, then look for instances of RHSs of rules in the current design ("analyze" the current design). Replace one by the corresponding LHS, by reversing the rule, and either try another implementation of the non-terminal or look for another instance of a RHS. By evaluating the design at each stage according to some optimization criteria, the system can choose the best alternative from among the possible implementations. Here again, control of search is an issue that should be handled separately.

As an example of this technique, consider the MUX example mentioned at the end of the previous subsection. This is where the $a0$ input is tied to 1. See Figure 3.14. The design has proceeded through top-down techniques to the point shown. Continuing with top-down techniques leads to a rather inefficient implementation that uses more transistors than necessary.

Note that the RHS of the BUFFER, Version 2, rule is isomorphic to the subgraph enclosed in dashed lines. Using that rule in reverse, the system produces the graph in Figure 3.15.

As an aside, the rule just used may seem to be useful only in one direction: that is, it is not clear why would one ever implement a BUFFER as AND $(1, x)$. It is not entirely implausible, however. One difference between this and a wire is

[6]One may easily convince oneself of this. Suppose it were not the case. Then after transformation there must exist an input vector to the whole device that results in some output being different than the original behavior. But this must be because some output of the subgraph lies outside the value-set of the induced role, else the change couldn't have affected the output of the circuit. But the rule application was assumed to be allowable, a contradiction.

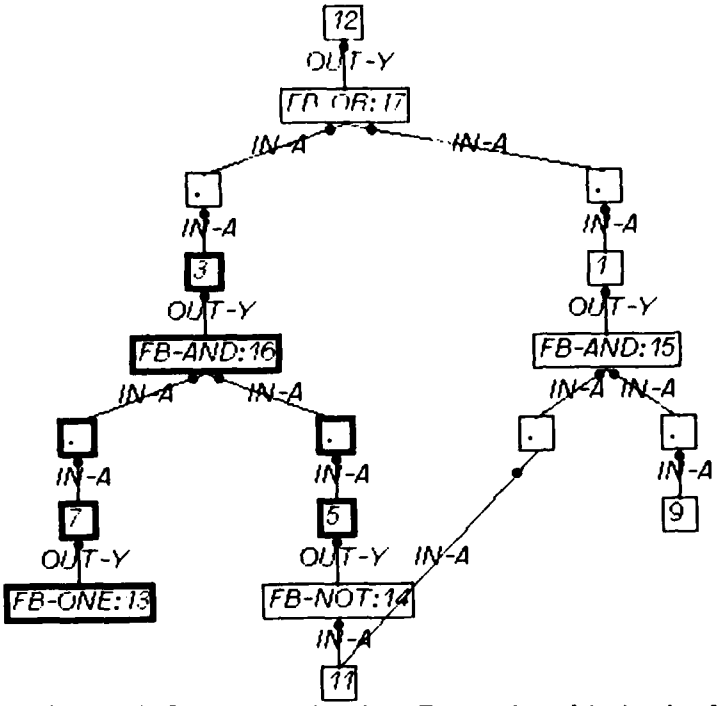


Figure 3.14: Initial Graph For Optimization Example: this is the MUX-ONE example after expanding the MUX by version 1. The highlighted nodes are found by the system to be isomorphic to the RHS of a BUFFER rule.

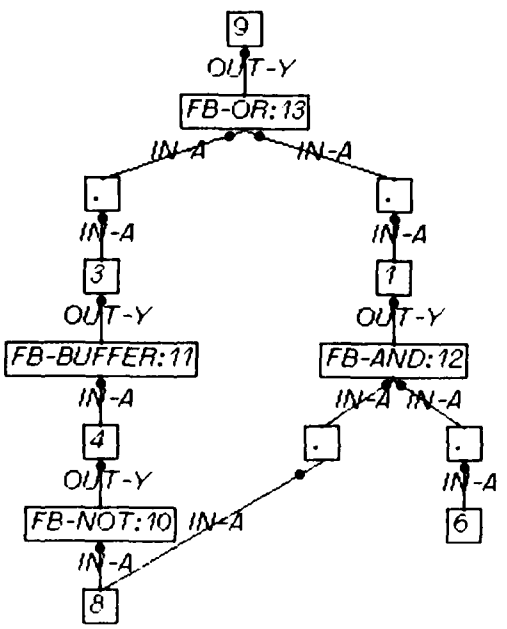


Figure 3.15: After Using BUFFER, Version 2, in reverse.

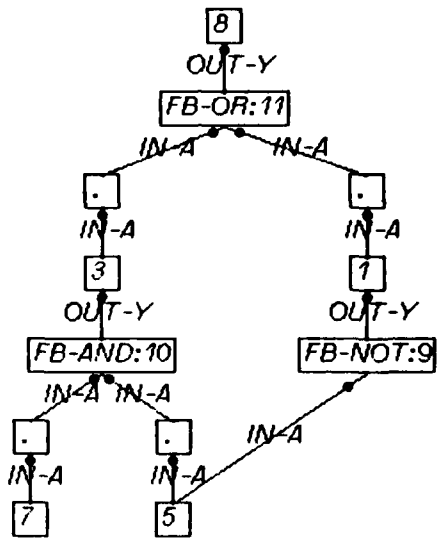


Figure 3.16: After BUFFER, Version 5, in forward direction.

that this is active; it amplifies the input signal to produce a more strongly driven output signal. Other reasons for choosing this implementation could be that the user doesn't know another active implementation, other implementations aren't possible given resource restrictions (as in gate array or standard cell methods of design), or it makes a layout more regular to have all AND gates present. Granted, rules may have strongly preferred directions of use, and some may even never be used in one direction, but there are many useful rules which are used in both directions.

Returning to the example, Version 5 is a better implementation than Version 2 if we are optimizing for transistor count. So the system may apply the BUFFER, Version 5, rule in the forward direction to get Figure 3.16.

Further optimization might be possible, but note that already the "direct" implementation, that obtained by choosing all terminal graphs, would contain 4 fewer transistors than would the direct implementation of the initial graph (Figure 3.14).

Determining Allowability

There is this problem of deciding when a given rule application is allowable. The currently implemented system deals with this crudely by just assuming that every possible rule application is allowable, except backwards applications of rules whose RHSs are terminal graphs (*i.e.* once it decides on a terminal implementation of some block it never changes its mind).

The general way of handling this, that requiring the most semantic knowledge, is to have a behavioral analyzer that can perform some reasoning based on the semantics of the elements and decide whether the substitution would affect the overall behavior.

The justification for not taking this approach is methodological and empirical. First, the system may not have the knowledge. It is beyond the scope of this thesis to study acquisition of the type of semantic knowledge required to perform behavioral reasoning. Second, almost all of the rules the system has used have been equivalence-preserving. Also, there haven't been too many rules with a given LHS or RHS, so just trying each out and testing the result is not too costly. So for practical reasons, it hasn't been necessary to attempt to determine allowability on a case by case basis. When this method is applied later to larger scale examples it will probably be necessary to deal with this problem, because highly optimized designs tend to capitalize on "don't cares" in the circuit, etc.

It may be that a stringent system of value types imposed on the connection points in the graphs will alleviate this problem; that way the system wouldn't keep trying, for example, positive-logic (TRUE = 1) implementations for negative-logic (TRUE = 0) behaviors. It might even be possible to introduce a type hierarchy, so that things of type twos-complement integer can match things of type integer, and so on. This probably won't completely remedy the problem, but it may render it small enough to handle by extra search.

Graph Matching

In the foregoing example, the system kept "looking for instances of the RHS" of various rules. This is, of course, a non-trivial problem, because arbitrary graph structures can be encoded in these graphs and the *Subgraph Isomorphism Problem* is known to be *NP*-complete[4].

The subgraph isomorphism problem can be stated simply: given two graphs, G1 and G2, determine whether G1 is isomorphic to a subgraph of G2.

The task this system faces would appear to be even harder: not only decide if one is isomorphic to a subgraph of the other, but find the matchings if they exist. That is, find the functions f mapping vertices of one into vertices of the other so that f is a graph isomorphism.

The bad news is that, yes, *in the worst case* the problem can be shown to require exponential time, because there are cases where a graph is isomorphic to a subgraph of another in exponentially many ways[7]. Even if the small graph is not isomorphic to a subgraph of the big graph, it can require an exponentially large amount of time to determine that, assuming $P \neq NP$.

The good news, however, is that there are domain features constraining the graphs that the system encounters. Nodes have types, and we don't allow matches that associate two nodes of different types. Arcs have types, so for a pair of neighboring nodes to match two nodes in the big graph, the pair in the small graph can be connected by an arc of a given type if and only if the pair in the large graph is. By encoding domain constraints like "two outputs are never connected to the same variable" in the graphs as arc/node type conventions, constraint can be added to

[7] Consider, for example, how many ways the complete graph on m vertices is isomorphic to a subgraph of the complete graph on n vertices for $m = n$, n"

the matching process. This is true only if the matching algorithm can take them into account, of course.

One result of this research is just such an algorithm. See the Appendix for a more detailed description of how this works. In a nutshell, it uses a local constraint propagation technique to throw away candidate matches that are obviously bad. The method is reminiscent of (and inspired by) the line labeling algorithm employed by the Waltz and Huffman-Clowes line-drawing recognition programs. In fact it is a direct generalization of those techniques. It is similar to a number of other approaches to the general problem of finding consistent labelings [6]. See the Appendix for the relation to other work.

3.3.3 Analysis

Analysis problems are hard. Specifically, problems of the form, "Show that X implements Y," are essentially problems in *parsing*, finding a derivation of X in grammar rules starting from Y. In Appendix A the recognition problem for arbitrary Design Grammars is shown to be uncomputable.

It follows from the uncomputability of recognition, that even if it is guaranteed that the graph is derivable from the grammar, there can be no algorithm that finds the derivation of it in a time bounded by a total computable function. There will always be cases on which a given algorithm either fails or takes impossibly long[8].

To see this, suppose there is an algorithm A, which, assuming the input actually is a derivable graph, is guaranteed to find a derivation of it in no more than $f(n)$ steps, where n is the size of the graph to be tested. Well, then we could *recognize* derivable graphs by simply running A on any candidate, t, and simultaneously counting the steps A uses. If A ever reaches $f(|\ t\ |)$ steps, then halt in failure, because if t were derivable, A would have stopped by now. But we already know that recognition is uncomputable, so by contradiction, we find that A can't exist.

Thus, the best one may hope for is an algorithm that can analyze designs in many cases. It is this perspective that influenced the design of the analysis algorithm to be given here.

The method is essentially a top-down parsing method: we start from the high-level graph and apply rules to it, searching for a way to transform it into the other graph. It may be viewed as a "greedy" method, in that it seizes upon the first sequence of operations that seems to make progress, and it never tries an operation that would undo any of the progress. The algorithm uses breadth-first search with heuristic pruning: it prunes those sequences which have yielded no progress after a given number of steps from the current start node[9]. It evaluates each state resulting from an operator application for progress. If no progress is made by any sequence of operators out to the search depth, then the program gives up. If progress is ever found, all searching is stopped and the state in which progress was made is taken

[8] A British Museum algorithm will certainly work every time, but its time complexity is not bounded by any total computable function.

[9] This given number, the "depth," is a parameter which may be changed. Appendix E shows examples which indicate the effect of changing this parameter's value.

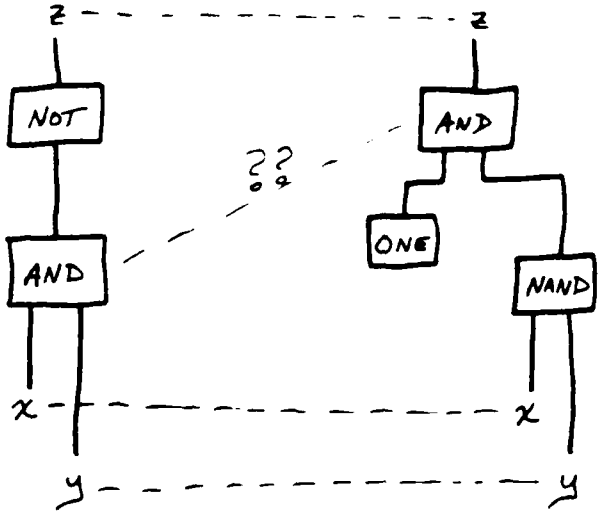


Figure 3.17: An Incorrect Matching of AND Boxes: the AND boxes do not correspond functionally, as the right one drives the output z and the left one can not possibly do so (z is NOT of its output).

as a new starting point from which to search. The progress is recorded, however, and no search path will be considered that requires altering a portion of the graph that already matches the target.

The algorithm measures progress by keeping track of how much the current state and the target look alike, and if this increases, then progress is made. As the target is fixed, and the size of the common subgraph increases, the algorithm will eventually consume the entire target graph and halt in success, unless after searching a fixed depth it found no progress. Then it halts in failure. Hence, the algorithm always halts. Moreover, it only keeps going as long as it makes some progress.

Looking Alike and Criterion R

The rest of this subsection describes the algorithm's method of deciding when two subgraphs "look" enough "alike" to merit associating them. Looking alike can be taken to be partial graph isomorphism: the program incrementally builds up a matching between the two graphs. This problem is a very interesting one: it would be useless simply to take any partial matching between the two. The simple reason is that even though two subgraphs match, it may be that the match won't extend to a total match, even after other derivations. An AND gate across two of the circuit inputs won't match one whose output is a circuit output, unless the entire circuit is an AND gate. Thus, matching the AND subgraphs would be incorrect. See Figure 3.17.

Note, however, that if both AND gates were across corresponding circuit inputs, or indeed any set of inputs already matched, then it would almost certainly

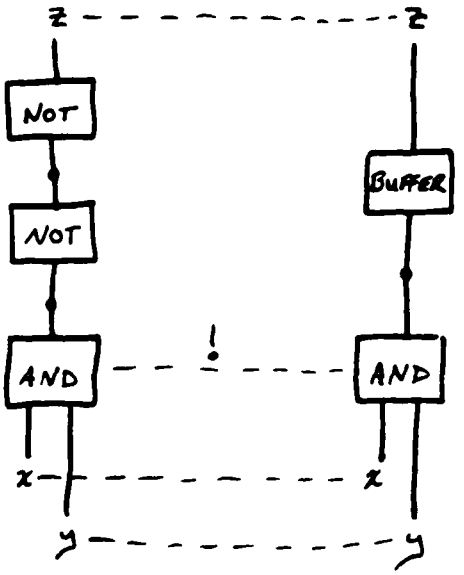


Figure 3.18: A Correct Matching of AND Boxes: these appear (syntactically) to compute the same subfunction of the overall circuit.

be correct to match them. See Figure 3.18. This fallible assumption is based on the following reasoning. We first assume that any nodes already matched are behaviorally equivalent. Because the inputs to the ANDs correspond, the outputs of the ANDs must be equivalent behaviorally. Thus, if the circuit outputs are equivalent and depend on the value computed by the outputs, then the functions computed between the outputs of the ANDs and the circuit outputs must be equivalent. Therefore, the grammar should be able to derive that equivalence. So the program might as well associate the AND outputs.

This must certainly be true if *the role induced by the unmatched portion of the graph is a behavior.* Call this CRITERION R. The partial matching algorithm can only fail when Criterion R fails to hold[10]. If the induced role is a behavior, then it must certainly match precisely the behavior of the unmatched portion, else there would be some combination of input values that forced an incorrect value on the outputs of the unmatched subgraph, causing an incorrect value on the outputs of the entire graph. Criterion R is a sufficient condition for success, but not necessary.

Consider, for example, a circuit with some inputs tied to constants. It is always the case that the role induced by the complement of a constant function is not a behavior: because the input to the complement corresponding to the constant never sees the other values, its behavior on those forbidden values is not determined by the overall behavior. One could associate any pair of constant-functions (with the same value), as all are functionally equivalent, but it would be erroneous to assume

[10] Or when one or more of the rule applications which led to the current situation was unallowable. This implies that the two graphs are no longer behaviorally equivalent overall. For the purposes of this discussion I will assume that the rule applications are allowable, since determining allowability is an issue which may be treated separately. Failure of Criterion R, on the other hand, is an issue which can't be avoided when using this algorithm.

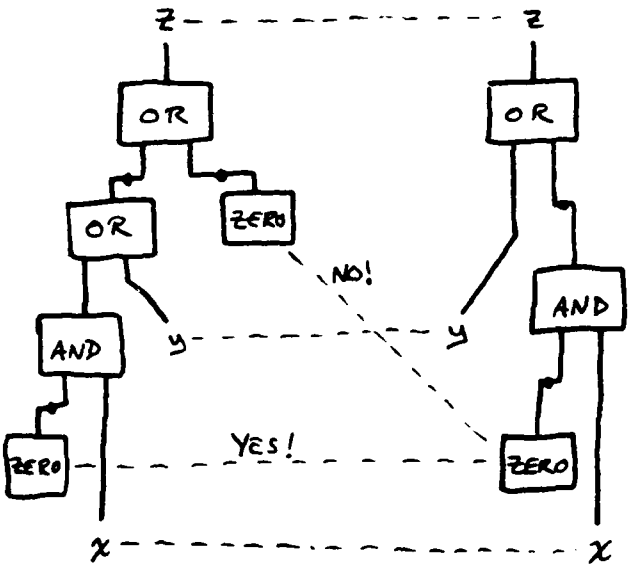


Figure 3.19: An Example With Confusing Possible Associations: which left-hand ZERO should the system match with the right hand ZERO, if any?

that the subgraphs they are input to are functionally equivalent; they could have different values on inputs not allowed by the constants. See Figure 3.19 for an example where some associations of ZERO boxes are correct, and others aren't. Of course, this generalizes to more than just roles induced by constant functions.

Another potential failure is when two nodes in the target graph are driven by isomorphic input graphs. That is, associating to one might be erroneous, because it actually should have been associated to the other. This is solved by computing equivalence classes of nodes of the target graph and associating a node in the problem graph to an equivalence class of nodes in the target graph, rather than any single node in the target graph.

The other method of associating nodes in the two graphs is looking for cases of invertible functions driving associated (matched) outputs. "Invertible" is defined heuristically as any time some output of a function is associated and all but one input is associated[11]. A better term for this might be *pseudo-invertible*. See Figure 3.20 for an example.

This may only fail if Criterion R fails to hold. A simple example of its failure is when we associate the ANDs in the two expressions "AND (f(a), 0)" and "AND (g(a), 0)." f and g may clearly be any functions, not necessarily equivalent.

In summary, the matching creeps in from the input and output edges of the graphs.

[11] Actually, we can look for the even weaker condition of all but one input of a given input-type being associated. Consider the MUX, for example, which has distinguishable inputs: we would consider it invertible even if no inputs (all but one, for each input type!) were associated.

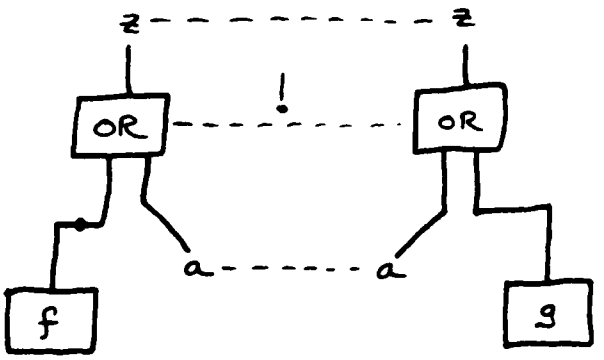


Figure 3.20: Associating Pseudo-Invertible Functions: as z and a have matches, and OR only has one other input, it is reasonable to assume that the ORs (and hence the outputs of f and g) should be matched.

Another Progress Criterion

There is another way in which a transformation can bring about a graph which looks more like the target graph than the source graph. If there exists a node whose inverse image under the partial match decreases in size because of the transformation, then that is deemed to have increased the quality of the partial match.

This obscure condition is important for parsing many optimized designs. For example, consider the pair of functions: y = BUFFER (NOT (x)) and y = NOT (x). The algorithm would match the right-hand y to both the left-hand y and the connection point between the BUFFER and the NOT[12]. Now, if the BUFFER - (connection point) transformation were applied subsequently, no progress would be observed without the inverse-image condition! With this condition, however, the right-hand y would have its inverse image decrease in size from two to one, so the system would judge that progress was made.

3.3.4 Analogical Design

Analogical Design is the process of using a known derivation of the solution to a problem to guide the search for a solution to a similar new problem. Either a teacher can give the program the derivations of the precedents, or the system can find them by analysis, if they're not too difficult for the analysis program to figure out. Two questions are addressed in this subsection: How and Why.

How

Analogy is such a general technique that it can be applied to all types of knowledge. The method used here applies it to structure function knowledge. It is not meant

[12]It would also match the NOT boxes.

to be all-inclusive of methods of Analogical Design.

The strategy is straightforward, given the Design Grammar formalism. Suppose the problem solver has a grammar derivation of a previously solved problem. This is a derivation that starts from a high-level graph and proceeds via rules to a low-level implementation. The problem solver is then posed a new problem in the form of a high-level specification.

The analogical method presented here looks for the best *partial* match between the problem graph and the precedent high-level graph. A partial match is an isomorphism of a subgraph of one to a subgraph of the other. "Best" is temporarily defined as "one involving the most nodes." This definition is expedient, and other approaches, like importance-dominated matching[22], might be more fruitful.

Once a partial match has been decided upon, the system proceeds to follow the sequence of transformations used on the precedent. For any given rule application, if the subgraph replaced in the precedent is contained in the domain of the partial match[13], then the transformation is performed on the problem graph. The partial match is updated to reflect the change; all nodes spliced into the problem graph are matched to the corresponding nodes spliced into the precedent graph. This way, sequences of expansions can be applied. Figure 3.21 illustrates this technique.

If, on the other hand, some node in the to-be-replaced subgraph of the precedent graph has no match in the problem graph, then the rule is not used on the problem graph. The partial match is pruned by eliminating all nodes in the to-be-replaced subgraph. Thus, every inapplicable operator tends to reduce the size of the partial match. The process is finished either when the partial match becomes empty or at the end of the precedent's operator sequence. Figure 3.22 illustrates.

The process of following a sequence of precedent operations is fast, because there is no search involved. The implemented system has performed analogical design on a 46 step derivation, 35 of the steps being applicable, in only the time required to apply the steps without search.

This method for using precedents is quite similar to the way they are used in Steinberg and Mitchell's REDESIGN system [19]. In that system, however, the authors supplied the partial matching by telling the system just what was different between the precedent and problem.

Why

One reason to use precedents this way is that it is one way to approach the search control issue: cut down search by restricting attention to search paths "near" ones known to have worked in the past. This is of course a heuristic notion, but with much support in everyday experience.

It is a nearly painless way of traveling far down a search tree to increase lookahead, with a fair chance of being on the right track. It is nearly painless because the algorithm given requires no search to follow a derivation, hence can do it fast. The only complex operation might be the partial matching at the beginning, but because initial problem specifications tend to be high-level they tend to be small

[1] that is, every node is matched to some node in the problem graph

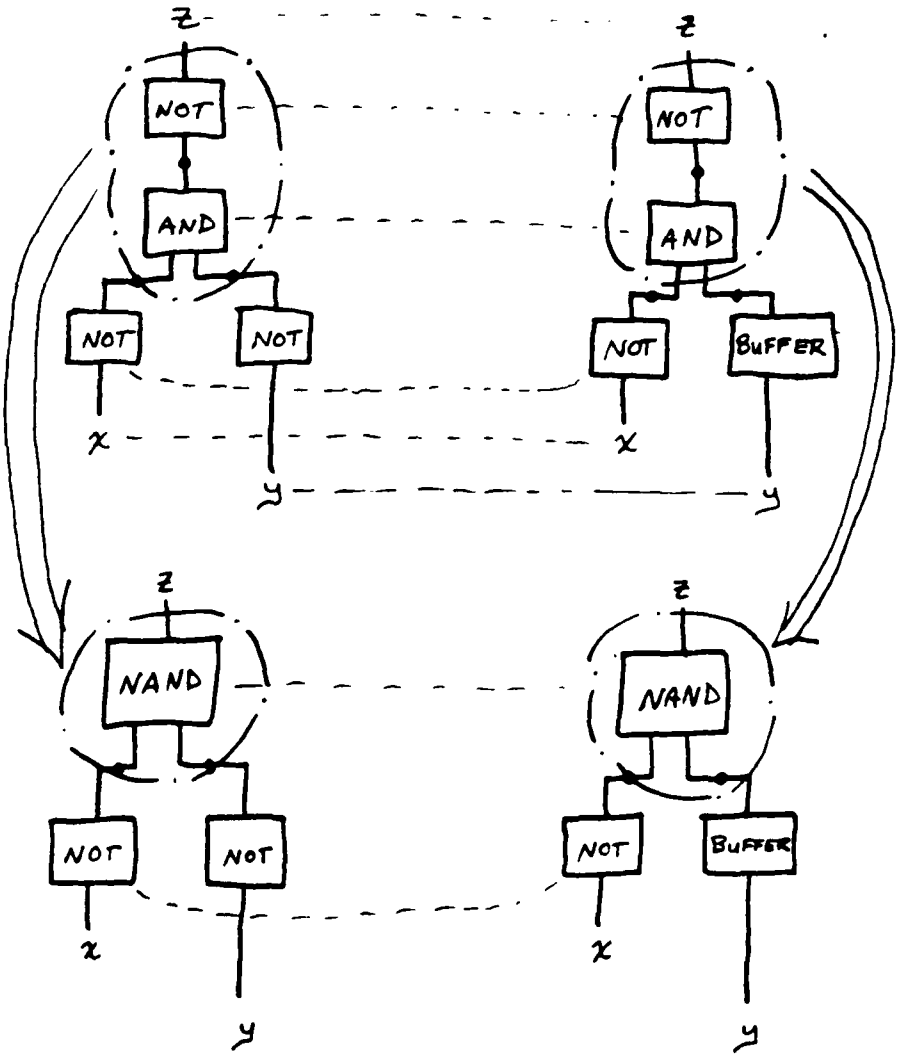


Figure 3.21: A Successful Analogical Rule Application: as the RHS of the NAND transformation, which was applied in the precedent derivation (left graph), matches a subgraph which is completely contained in the domain of the partial match (dashed lines), the system applies the same transformation to the problem graph (right graph).

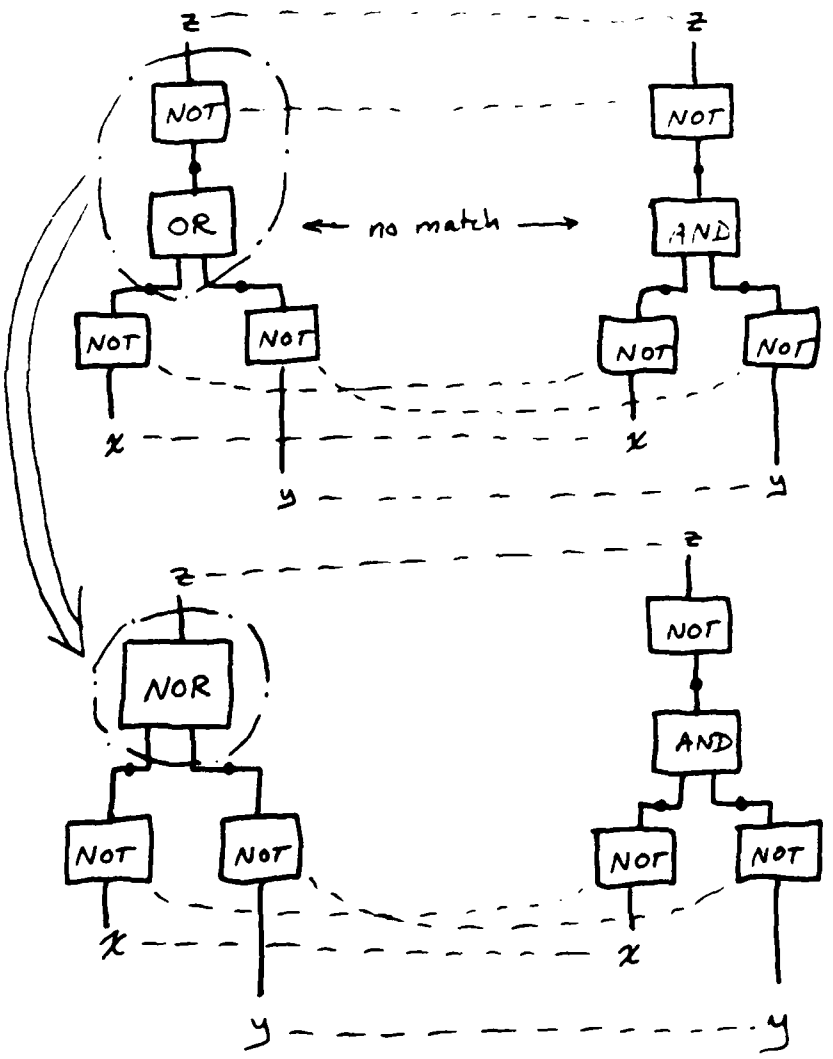


Figure 3.22: A Failed Analogical Rule Application: the domain of the NOR transformation is *not* contained in the domain of the partial match, because the OR does not match anything.

graphs. Small things are always easy to match. It is only the *implementations* that get large, and the method doesn't require partial matching after the beginning.

Another advantage of this technique is that one can effectively store and access a large amount of information with a relatively low cost. Because there are potentially exponentially many partial matches of the initial graph, there could be exponentially many different outcomes of applying Analogical Design to the precedent. That is, for any partial match of problem to initial graph, a certain subset of the operations in the sequence will remain applicable. Thus, as there are many subsets of the nodes of the initial graph, there are many potentially useful subsets of the operator sequence (they need not be subsequences; that is, they need not be contiguous). Thus, a single precedent encodes exponentially many explained Design Grammar rules. Accessing these is relatively easy as explained above. This is much more space-efficient than having all those rules explicitly represented in the grammar[14]. Especially because it is quite plausible that only a small portion of those rules is actually useful.

It may seem strange to argue that something is good because it encodes many rules, most of which are useless. The problem is that it is difficult to know in advance which are useful and which aren't. Thus, rather than storing many useless rules in order to be sure to have the useful rules, we store one precedent. (It may still be a win, even if the useful rules are known in advance, as the system still has to store and use more than one, instead of one.)

This sort of Analogical Design is not a new idea. As mentioned above, Steinberg and Mitchell explored it. Winston '21 uses this technique in reasoning about function. The MACROPS idea in STRIPS 3 can also be viewed as an implementation of Analogical Design in a state operator formalism that far predates (1972) Mitchell and Steinberg's use. There are significant differences between the STRIPS triangle table approach and Analogical Design. One is that STRIPS's formalism is based on a sequential, state-space model which imposes a total order on the precedent's operation sequence. This is not the most natural model for design, because transformations of separated parts of the current graph are only loosely coupled if at all. Also, non-contiguous subsequences are found naturally with Analogical Design, while the STRIPS system would seem to favor only contiguous subsequences. It appears that it would take extra effort to get a non-contiguous subsequence out of a triangle table than out of an Analogical Design precedent.

3.3.5 Some Desirable Properties a Design Grammar Should Have

In the next Chapter, I'll deal with how the system learns Design Grammar rules from example designs. But first, it is necessary to understand what properties the DG should have in order to maximize power and efficiency.

For example, if the system simply stored a precedent, it could be said to "learn"

[14] It is probably faster, too, at least on serial architectures. Consider the problem of accessing a large database of rules versus using this method once.

in that it could then solve a new problem: precisely the one solved by the precedent. This wouldn't be too bad from an efficiency-of-use standpoint. After all, there would only be a small number of rules which probably wouldn't combine very well, so virtually no search would be possible in design, optimization, or analysis; and analogical design precedents would all be one rule long.

Obviously, however, the system would not be very powerful, either. We could ask that the teacher be kinder and only give precedents that are generally useful rules and store these. This would be nice in a perfect world, with a perfect teacher. But this is just what knowledge engineers try to get experts to do in programming an expert system. The problems with this technique, also known as the knowledge bottleneck, are well documented: it is hard to get experts to agree on which rules are useful, it is hard to keep the rules consistent, it is hard to keep out redundancy and its concomitant inefficiency, and it is hard to know when the expert is finished (the rules are complete).

So what is needed is a technique whereby the system can help organize the database itself and learn from complex precedents, thereby taking some pressure off of the teacher. Separating rules into derived and primitive rules can help with this.

- If a rule is derivable in terms of other rules, then it is definitely consistent with those rules. If an inconsistency is found, those rules derived from the bad rule are known, so they can be reexamined. The others don't need to be checked, because they are independently justified.

- If some rule is "an obvious consequence" of some rules, then it can be thrown out of the database, thereby helping to alleviate the canonical expert system problem: becoming so smart it takes forever to do anything. What "obvious" means depends on the task and the algorithms, but intuitively it means that one rule is a short derivation from one or more others. After all, the only reason to keep derived rules at all is to cut down search by having common search paths made explicit. If the path is short, why not save space and just regenerate it every time it's needed? Another reason to throw away a derived rule is if occasions of its use don't arise very often. This is an instance of *The Lemma Problem*: which derivable facts should be kept for efficiency, and which should be thrown out for efficiency? To my knowledge there hasn't been much progress made on this problem, but it seems clear that *some* derived rules should be discarded.

- Analogical Design as I've defined it can only be used with derived rules or precedents. Thus, the system gains more power to do this type of problem solving if it can derive more equivalences.

- Analysis is obviously made easier by having general rules that explain many equivalences. Also, having explained lemmata (macros) around can help speed up the search involved in Analysis.

- Some primitive transformations are so small that they virtually never occur, except embedded in one particular sequence. The sequence, however, occurs frequently. Then, too, it would make sense to keep the sequence around for efficiency. It would not, however, make sense to throw away the primitive rule; there are those rare cases where it is used differently. Contrast this with the case of a little-used derived rule.

The common theme in these observations is that it is better to derive most of the equivalences that the system knows. Therefore, **one goal of the Learning system is to acquire as general a set of primitive rules as possible.**

Another consideration, which is related more to efficiency than power, is the issue of recognizing instances of RHSs as subgraphs. Even though the graph matching algorithm seems to be fairly efficient, it still takes much longer to match large graphs than small ones. So the *NP-completeness of subgraph matching* leads to the second learning goal: **the RHS graphs of the rules should be small.** Size is measured by the number of nodes in the graph. Thus, representation is an issue here as always: it may be worthwhile to create a macro symbol for a useful subgraph simply to speed up the matching process.

The final interesting thing to note is that these goals are compatible. Primitive rules tend to be smaller than rules they help derive because they tend to be subgraphs of them.

3.4 Summary

A Design Grammar is a graph grammar whose rules encode knowledge about structure and function. That is, the LHS represents a functional block and the RHS represents one decomposition (implementation) of that function. Terminal graphs are composed of directly implementable elements and combination operations. A behavior is a function from an input space to an output space, while a role is a mapping from the input space into the power set of the output space. The induced role of a subdevice is the least constrained role that the subdevice must satisfy to preserve overall behavior of the device. Equivalence-preserving rules are those for which the behavior of the LHS is identical to the behavior of the RHS. Equivalence-preserving rules may be used in either direction. More generally, any time the role induced by a recognized rule-side is satisfied by the other side of the rule, the rule application is allowable. A rule is recorded as either primitive or derived, according to whether it has a known derivation in terms of other rules. There are a few generic transformations that every system should have available.

Top-Down Design arises from the system using grammar rules in the forward direction. Optimization arises when rules may be used in all allowable directions. Analysis is a matter of parsing. Analogical Design can be achieved by creating a partial match between high-level graphs of the problem and a derived precedent, then re-running the derivation, throwing out the steps not covered by the partial match.

Chapter 4

Learning by Failing to Explain: Precedent Analysis

This chapter and the next will discuss how a Design Grammar is learned from example designs by the system which implements Learning by Failing to Explain. This Chapter discusses how a new rule is conjectured from a precedent. I call this *Precedent Analysis*. The next Chapter discusses what to do once the conjecture is made. The first section discusses what a precedent is, and why learning from precedents is not only a matter of generalization. The second section presents an algorithm that exhibits this behavior. The third section demonstrates how the system can conjecture rules that are not equivalence preserving. The fourth section deals with learning rules whose graphs contain parameterized elements.

4.1 Precedents As Complexes of Examples

A typical input to the learner will be a precedent, by which I mean a pair of descriptions of the same device, together with the correspondence between the inputs and outputs. That is, the system needn't guess what the variables are or which implement which. Either description can be thought of as an implementation of the other. However, because of the way the algorithm is implemented, it is convenient to think of one graph as being "more high-level" than the other. I will refer to one graph as the *high level graph* and the other as the *low-level graph*. See Figure 4.1 for an example.

From some set of precedents, the learner is to build up a Design Grammar that generates at least the precedents seen. It should also, however, strive toward the goals of maximum power and efficiency, as discussed in the previous chapter.

One key idea for attacking this problem is to *assume that the precedents are constructed from Design Grammar rules*[1]. Hence, each precedent is a combination of instances of several concepts. For example, the precedent in Figure 4.1 consists

[1] This is no loss of generality, because every precedent could be taken as a pair of rules by creating a non-terminal with each graph as an implementation. Granted this wouldn't be very powerful, but it certainly covers every conceivable case.

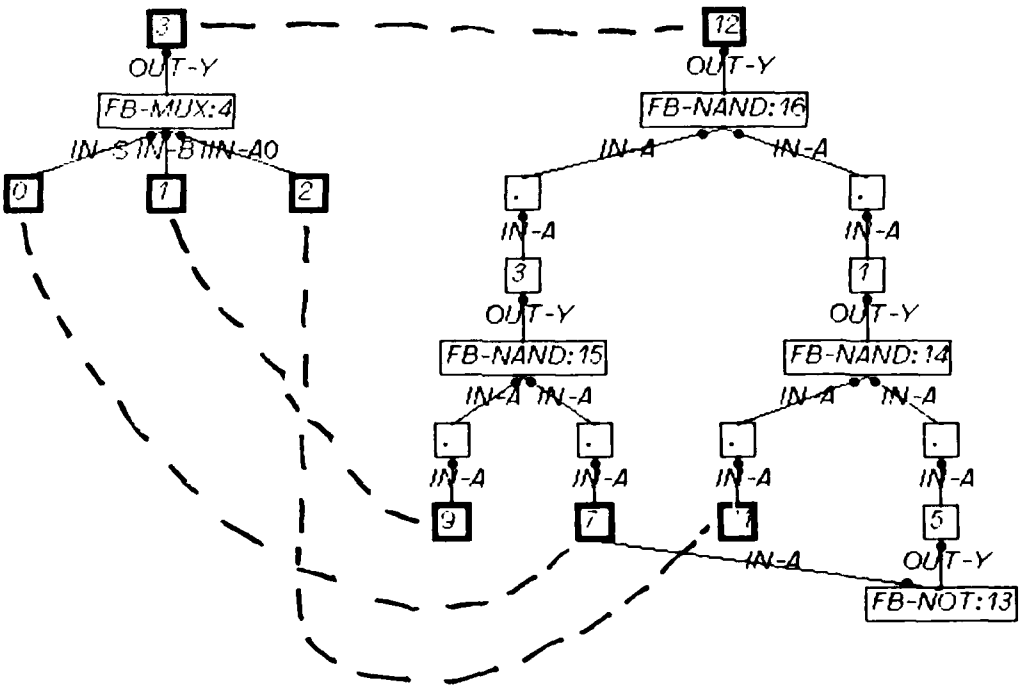


Figure 4.1: A Precedent: dashed lines indicate correspondences given to the system with the precedent.

of[2]

1. An instance of NAND, Version 1, down. (applied to device 1)
2. An instance of NAND, Version 1, down. (applied to device 2)
3. An instance of NAND, Version 1, down. (applied to device 3)
4. An instance of BUFFER, Version 6, up. (two successive NOTs)
5. An instance of BUFFER, Version 6, up. (two successive NOTs)
6. An instance of BUFFER, Version 5, down.
7. An instance of BUFFER, Version 5, down.
8. An instance of MUX, Version 1, up. (all of them together)

Note that multiple interpretations are sometimes possible. For example, NOT (NOT (NOT (x))) $\Rightarrow$ NOT (BUFFER (x)) $\Rightarrow$ NOT (x); or NOT (NOT (NOT (x))) $\Rightarrow$ BUFFER (NOT (x)) $\Rightarrow$ NOT (x).

This work is concerned with *learning many concepts from complexes of examples.* A complex of examples is a single thing that is composed of instances of many concepts, such that the boundaries between instances are not *a priori* clearly defined. Note the contrast between this and many Machine Learning studies: it is not trying to induce a *single* concept from a number of examples, *presented as instances of the concept.*

The difference between the two paradigms is that in addition to generalizing examples of a concept, the learner must first find the examples and group them into appropriate classes for generalization. This thesis will deal primarily with finding and grouping examples, although the implemented system also does some generalization in the Gear World.

4.2 The Algorithm

The key idea in this Learning algorithm is simply stated: **Use knowledge you already have to recognize which parts of the precedents are new.** The examples in this section will refer to the Design Grammar of Appendix D. Suppose the system has already acquired it somehow, possibly by this method[3].

4.2.1 What Happens

The teacher shows the system the precedent in Figure 4.2. The Z boxes represent the function whose output is equal to its input of the previous clock cycle. The

[2] See Appendix D for definitions of the rules used.

[3] The base case of this apparent recursion is that the system simply remembers the first precedent it sees.

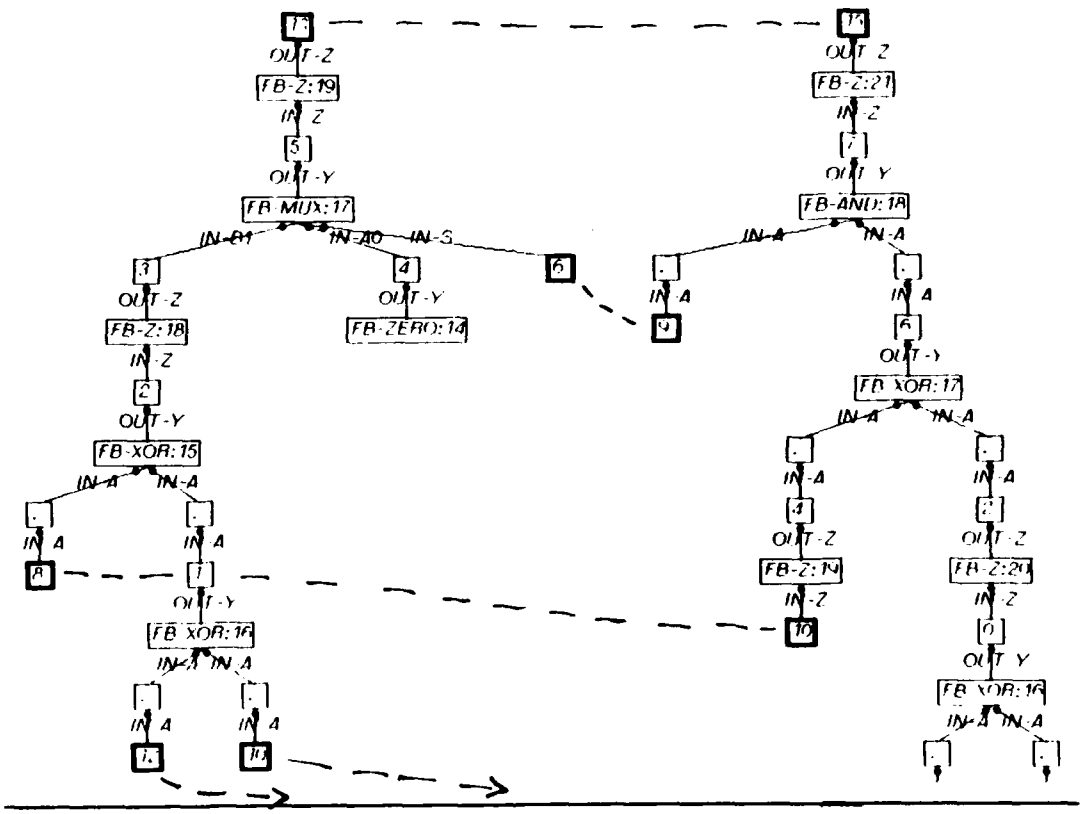


Figure 4.2: Learning Example Precedent: Z boxes are delays; the two input variables of the right graph are just off the bottom of the Figure.

High-Level-Graph is a straight-forward transcription of the requirement, "After any clock cycle, if K was 0 on the cycle before, then y is 0; else if K was 1, then output is the XOR of the values of a, b, and c two cycles ago." This is plausible as a description of a portion of the function of an ALU.

The Low-Level-Graph is an optimized version that is faster, because it allows a shorter clock cycle time. The two XORs together result in 6 gate delays between clocks, while putting the XORs on different clock cycles allows the cycle time to be as little as four gate delays (second XOR plus MUX) in this case.

One possible rule to glean from this is just the entire precedent. But this is rather conservative; it may be that a new, useful rule was used in addition to many that the learner already knew. It would be nice to find out what the new rule was; then the whole precedent would be derivable from a set of smaller, more general rules. This is the central idea behind *Precedent Analysis*: use what you know to help find the most general candidate rule.

The reasoning behind the algorithm is as follows. "Because I assume the designer used grammar rules to derive the precedent, there must exist a sequence of transformations that transforms the High-Level-Graph into the Low-Level-Graph. I'll try to generate that sequence from my own rules. If I can't quite make it, then what's leftover must be either a rule or a simpler complex of rules, because the unmatched subgraphs *induce the same roles.* Because the unmatched subgraphs induce the same roles, they are probably behaviorally equivalent, or at least they probably fill many of the same roles; therefore it is probably useful to remember the association."

Note that this problem is at least as hard, in general, as the recognition problem[4], because learning from a precedent implies being able to tell whether you already knew it. The trick here is to be able to generate a partial derivation that "gets close" to the real derivation before failing. But *it was precisely this consideration that drove the design of the Analysis algorithm.* (See Chapter 3.) The Analysis algorithm searches incrementally through sequences of grammar rules always making some progress, lest it halt. If it halts, then either it hasn't got the knowledge (*i.e.*, it's missing a grammar rule), or the sequence of transformations required to get it closer to the goal is too long.

The Learning algorithm calls the Analysis algorithm to do as much as it can, returning the partial derivation and the partial match it constructed. The system then conjectures that the two unmatched subgraphs, one from the transformed High-Level-Graph, one from the Low-Level-Graph, are functionally equivalent.

Note, however, that one of the stated goals for the Design Grammar is that the graphs be small. This is to avoid the problems of NP-completeness, among other reasons. Thus, it is desirable, before forming a rule, to *choose the smallest (in number of nodes) member of the class of graphs equivalent to those of the transformed graph.* The implemented analysis algorithm actually returns the smallest graph it examined before failure. That is, it returns the sequence of steps leading to the

[4] ...which is uncomputable. See Appendix A.

smallest graph which looks as much like the low level graph as possible[5]. This was used, for example, in the precedent demonstrated in the Precedent Analysis scenario in Chapter 2.

Figure 4.3 shows the transformed High-Level-Graph and the Low-Level-Graph, together with the partial match produced by the Analysis algorithm. The unmatched subgraphs are shown enclosed in dashed curves. The conjecture generated is that $Z(\text{XOR}\ (x, y))$ is equivalent to XOR $(Z\ (x), Z\ (y))$. [6]

This conjecture does not usually have the form of a Design Grammar rule: neither side need be a non-terminal graph. Call the unexplained subgraphs of the transformed High-Level-Graph and the Low-Level-Graph the extracted transformed High-Level-Graph and the extracted Low-Level-Graph respectively. If the extracted transformed High-Level-Graph is a non-terminal graph, then the system simply adds the equivalence as a new version of the non-terminal. (Recall that a non-terminal graph is a graph with a single non-terminal node, plus possibly some variable nodes connected.) If the extracted transformed High-Level-Graph is not a non-terminal graph, however, there are a number of options available, depending on the knowledge available to the system.

- It can ask the teacher to tell it the name of a non-terminal to use, and create two rules, one using the extracted transformed High-Level-Graph as RHS and the other using the extracted Low-Level-Graph as RHS.

- It can just make up a name on its own and create the two rules. Figure 4.4 shows how the system creates rules via this method.

- It can look through the grammar rules it knows, and try to find a previously known non-terminal that is equivalent to either the extracted transformed High-Level-Graph or the extracted Low-Level-Graph. It might do this by functional simulation, for example, or other functional reasoning.

The system can also, once a new rule is formulated, complete the original derivation and create new derived rules whose RHSs correspond to the precedent's two original graphs.

4.2.2 What Is Going On

In contrast to many learning algorithms, it is possible to get some insight into how this one can fail. To do this, we need to examine the assumptions underlying its operation. The process of generating a conjecture consists of creating a maximal partial parse, using the current knowledge base, and then extracting the unexplained part as a new equivalence to be made into rules. The Algorithm

[5] It maximizes "look-alike" before minimizing size.

[6] The Analysis algorithm expanded the MUX, optimized it because it has a constant tied to an input, and matched the resulting AND gate. The XOR gate across a and b was noticed and matched first.

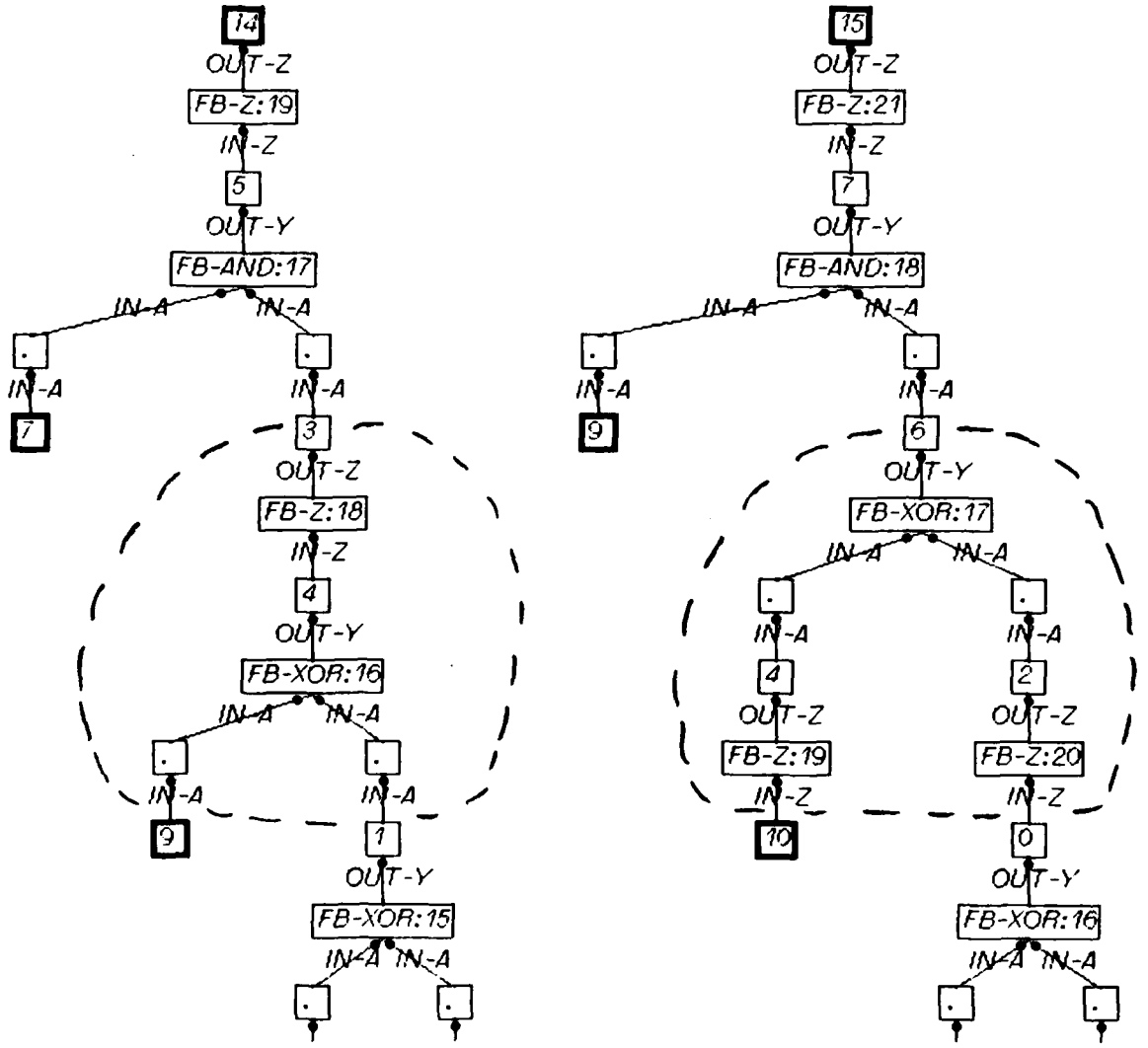


Figure 4.3: Learning Example After Transformation: subgraphs enclosed in dashes are only parts left unmatched.

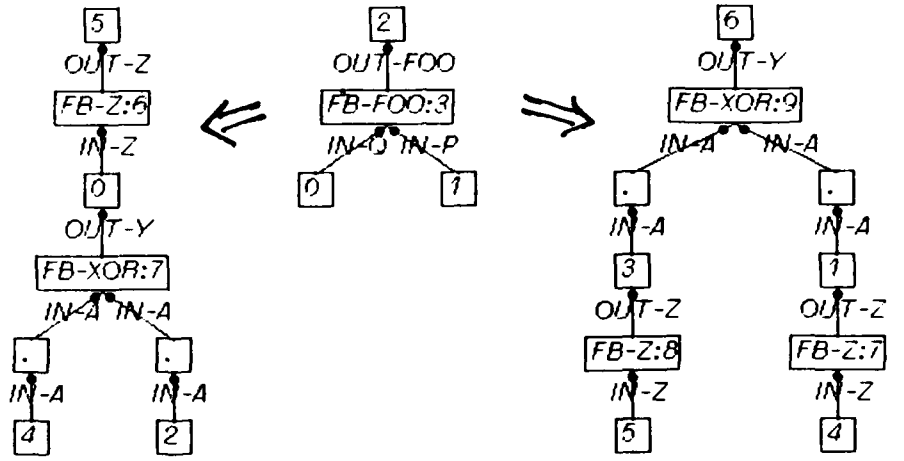


Figure 4.4: Creating Rules From Learned Associations: system creates new non-terminal symbol "FOO" and two rules; left graph is RHS of one, right graph is RHS of other.

- *assumes the partial parse generated by the Analysis algorithm is correct;* i.e. *that all node associations produced by it represent behavioral equivalences.* This can fail exactly when the Analysis algorithm can (see Chapter 3).

- *assumes that the conjectured equivalence will be true in general.* This is a form of analogy: because the two subgraphs fill the same induced role, they must be capable of filling many of the same roles. This is based on the intuitions that (a) design rules are generally useful, and (b) because the design was produced using design rules, and the matched portions of the graphs correspond to a set of rule applications, the design must "cleave" along the matched/unmatched boundary line. Notice that it does not depend on the unmatched portion being exactly one rule instance, just a constellation of instances whose boundaries all go up to the unmatched frontier and stop. The system later may be able to cleave the conjecture further, after it has more rules on hand, by re-analysis (see Chapter 5).

It is interesting to compare this with the learning aspect of Winston's ANALOGY program, as applied in the cup learning world 21 . ANALOGY accepts precedents passively and formulates conjectures about rules only when faced with a problem. To solve a given problem, ANALOGY matches precedents to it and finds causal structure that can be used to transfer knowledge. It then summarizes the causal chain which resulted in the transferred knowledge as a rule.

Casting this into the terms of the Learning system described here, ANALOGY accepts a derived precedent (it is given the explanatory links in the input), and

when it comes time to solve a problem, it applies what I've called Analogical Design (which is really just a technique for controlling the search for derivations) to the problem, using the store of derived precedents. It then collects all the derivation steps which were applied (as opposed to those which were skipped because they were inapplicable), and stores the initial graph and the final graph as a new derived rule, and gives an explanation based on the applicable derivation steps. ANALOGY learns only rules it can explain previously; in my jargon, it learns new derived rules. It was not designed to learn primitive rules.

We can gain another perspective on the algorithm by asking the question: given the truth of assumption one, that the partial parse returned by Analysis is correct, how is it possible for the conjecture to be wrong? Well, we need to understand what "wrong" means. It can certainly be "wrong" in the sense that the conjecture fails to represent a behavioral equivalence. The only guarantee is that the two graphs are role equivalent in the particular induced roles of the precedent graphs. But this reflects the real world of design: designers frequently take advantage of "don't-cares" in optimizing implementations.

An example of a useful everyday-life rule that does not preserve equivalence is using a screwdriver to pound tacks: a hammer is not behaviorally equivalent to the screwdriver because it is impossible to turn screws with a hammer, but both satisfy the role of tack-pounding implement.

On the other hand, the conjecture must definitely represent a role equivalence [7]. Therefore, it must represent some kind of rule. Moreover, it is a rule which has been used at least once, so it is more useful than the many possible conjectures which will never be used. It might be that the precedent is a completely special case, so that the conjecture will never be useful again. If so, the algorithm has done its job; it is the teacher who has failed by showing the system a pathological example.

For the same reasons as in the Analysis case, this sort of conjecture can only arise when Criterion R fails to hold[8]. That is, if Criterion R holds, then the conjecture must be equivalence preserving. The next section discusses an example of this phenomenon in more detail.

Another reason why the system appears to come up with useful rules is that in designed systems, all the objects and features are usually teleologically justifiable. That is, everything present has a relevant purpose. In other domains, this may not be the case. For example, in understanding a story there are often many irrelevant details present which have no clear explanation. Why does the main character have red hair? Well, why not? The effect this has on Precedent Analysis is that it would create overly specific rules, since it couldn't explain why someone had red hair.

---

[7] After all, given assumption one, the transformed High-Level-Graph is behaviorally equivalent to the Low-Level-Graph. This means the induced role of the unmatched portion of the transformed High-Level-Graph must be identical to the induced role of the unmatched portion of the Low-Level-Graph, because they arise from the syntactically identical, matched subgraphs. Hence, the extracted transformed High-Level-Graph must satisfy the same behavior as the extracted Low-Level-Graph. QED.

[8] Criterion R states that the induced role of a subgraph is a behavior. See Chapter 3.

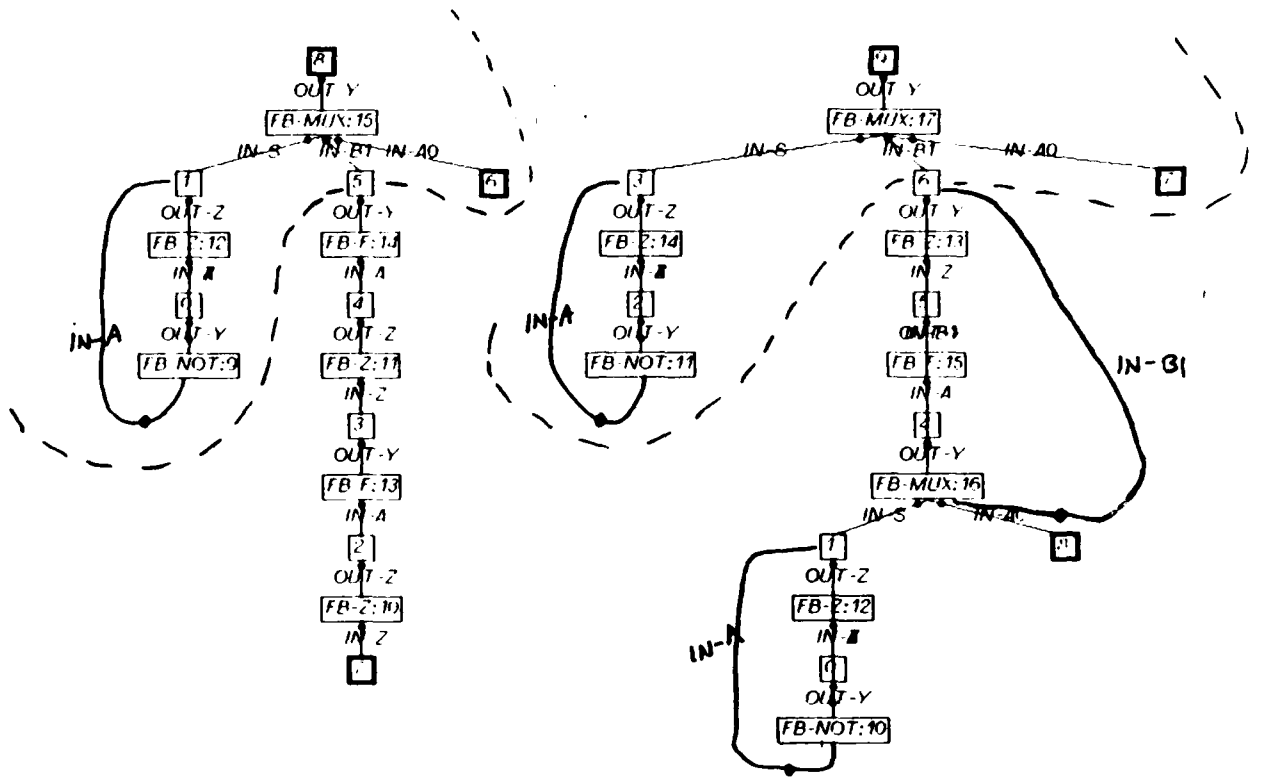


Figure 4.5: A Tricky Precedent: $f^2(a)$ with delays, read every other clock cycle.

4.3 A Failure?

This section gives an example where the system conjectures a non-equivalence-preserving rule. Consider the precedent in Figure 4.5. The High-Level-Graph is an implementation of the function $y(a, x)$ that is equal to x_t on odd clock cycles t, and $f(f(a_{t-2}))$ on even clock cycles t. The Low-Level-Graph is another implementation of the same function. The difference is that because the f is only really used on every other clock cycle, we can reuse the same box by feeding the partial result back to the input. Applying the analysis algorithm to this precedent associates the portions enclosed in dashes. The learning program then extracts the extracted transformed High-Level-Graph and extracted Low-Level-Graph shown in Figure 4.6.

The devices represented by these two graphs are certainly not equivalent. The extracted transformed High-Level-Graph puts out $f(f(a_{t-2}))$ on every cycle, while the extracted Low-Level-Graph puts out $f(f(a_{t-2}))$ on even cycles and $f(a_{t-1})$ on odd cycles. The reason this occurs is that the "rule" corresponding to this association is context dependent: if the external circuit samples its output only on the even cycles then it's true. If it uses the output on an odd cycle, then it's not.

As expected, Criterion R fails in the situation shown in Figure 4.5. The induced role of the unmatched part is the function shown below.

$$r(a) \quad \begin{cases} \{f(f(a_{t-2}))\}, & t \text{ even} \\ \{0,1\}, & t \text{ odd} \end{cases}$$

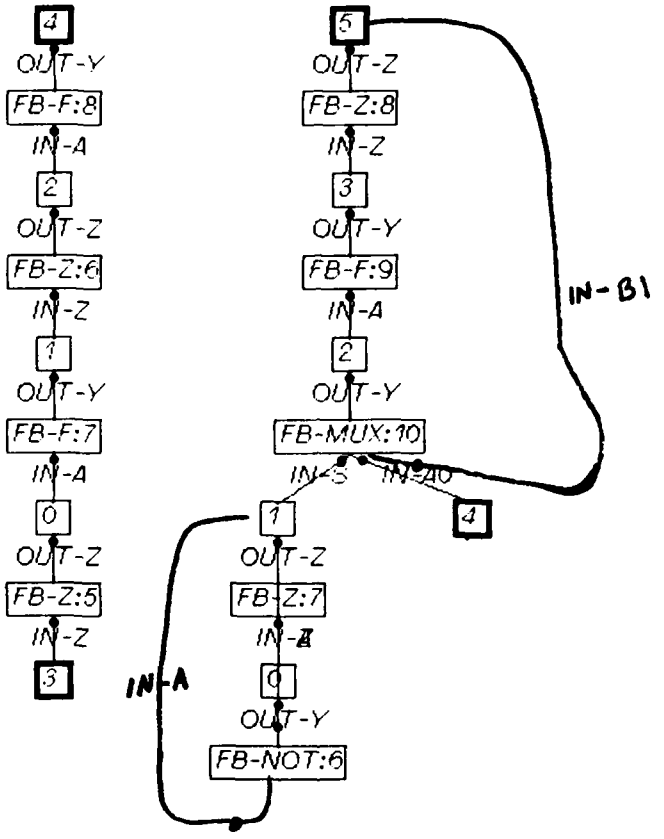


Figure 4.6: The Learned Association: this is only true in contexts where the output is used on odd clock cycles only.

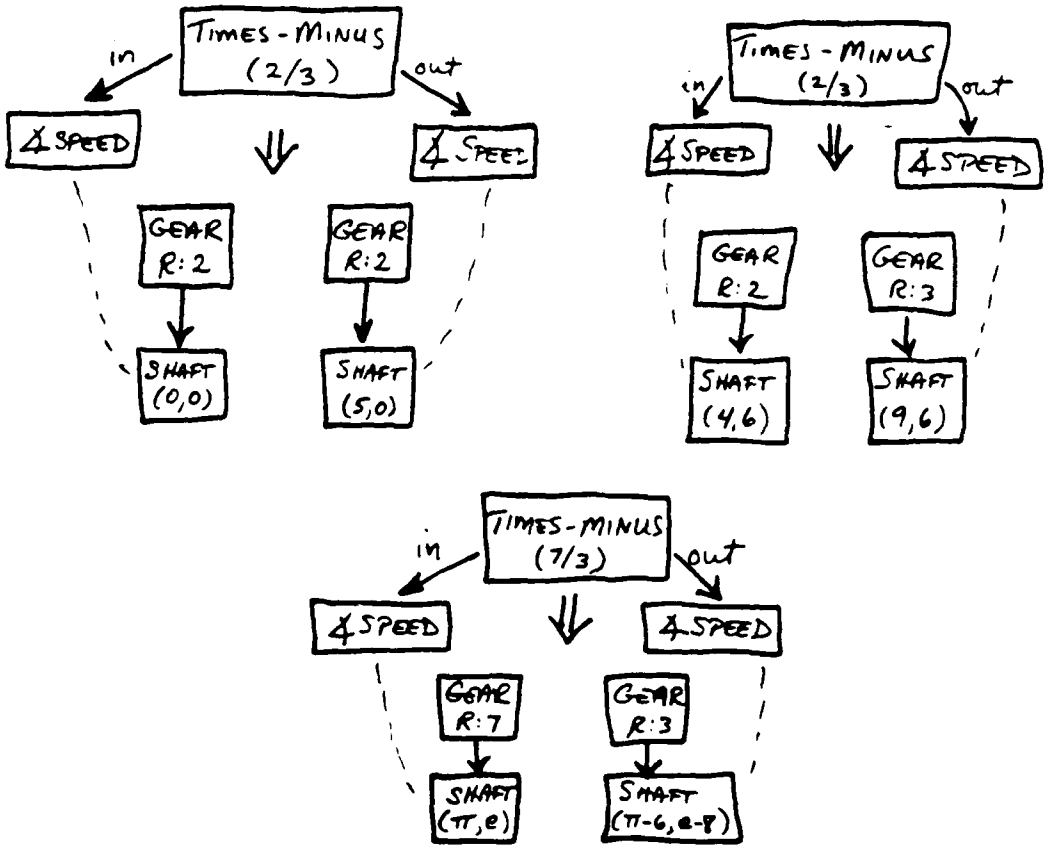


Figure 4.7: Different Rules? These represent specific gear mesh rules; differences may be described as parameter variations.

This is not a behavior, because it allows any value on odd clock cycles.

4.4 Parameters

There are domains where instead of having a small, finite number of structural operators (terminal elements), there are effectively infinite classes of them. The Gear World is one such domain. There are gears and sprockets of different radii, and elements have different spatial positions.

This creates special problems in representing grammar rules: it is obviously unsatisfactory to have infinitely many rules, one for each instantiation of the parameters of the elements making it up. See Figure 4.7.

One can represent these rules much more compactly in the style shown in the Appendix for TIMES-MINUS, Version 1. That is, represent the elements in a graph, their parameter values left indefinite, and record the relationships that must exist

between the parameter values for the rule's RHS to be matched. Note that the functional description elements (non-terminal elements) may also have parameters.

This, however, poses another learning problem: how can a learner acquire the parameter relations? The learning algorithm given so far is sufficient to extract particular *fully parameterized* rules, rules where every parameter value is specified with a constant. It does not as yet address the problem of the relations.

This is clearly a problem in inductive generalization, but with an added twist. Not only doesn't the learner know what the final form of the generalization is, but it does not even know which examples belong to which rules (concepts). This is the *example grouping problem* alluded to previously.

The program solves this problem as follows. Call the graph representation of a precedent with the parameter values left indefinite the *qualitative* component of the description. The program simply assumes that any two rules whose qualitative components are isomorphic are instances of the same concept. Both the (qualitative) structural descriptions and the (qualitative) functional descriptions must match correspondingly. This technique is not limited to rules where the RHS consists of terminal elements. It may be just as well applied to any rule. Thus, the three examples in Figure 4.7 are all qualitatively alike, so the system groups them together for generalization.

It may, as in the non-parameterized case, be necessary to create a non-terminal that stands for the two halves of the equivalence learned from a precedent. This happens just the same way, but the created non-terminal must have exactly as many parameters as the graph representing the functional implementation (the High-Level-Graph). So one rule created associates the new LHS to the High-Level-Graph, and the parameter relations are simply statements of equality; that is, each parameter of the LHS is equal to the appropriate parameter of the RHS. The other rule, the one corresponding to the Low-Level-Graph, is the one where the generalization takes place. See Figure 4.8 for an example.

This, too, is a form of analogy. The judgment of similarity is based on matching "up to parameters". Parameters provide a way of expressing a class of things that are closely related. This is the mark of a good representation: make things that are semantically closely related appear syntactically close. Differing in only parameter values is a very close match. Thus, the success of this approach to the grouping problem hinges (as do the other uses of analogy) on the degree to which the semantics of the domain is represented faithfully.

Whenever two or more examples of the same concept are found, they are given to a constructive generalization program[9]. The database is maintained in such a way as to keep the current generalization of the concept as the only version of the rule used by the system for analysis, etc., thereby reducing the sheer number of rules to search through.

The currently implemented system employs a crude constructive generalizer for

[9] *Constructive Generalization* is generalization where the program must not only climb a generality hierarchy, but it must create the generalized descriptions from lower level primitives as well. Having an *a priori* description language, in terms of which the concept is guaranteed to be expressed, is a powerful constraint.

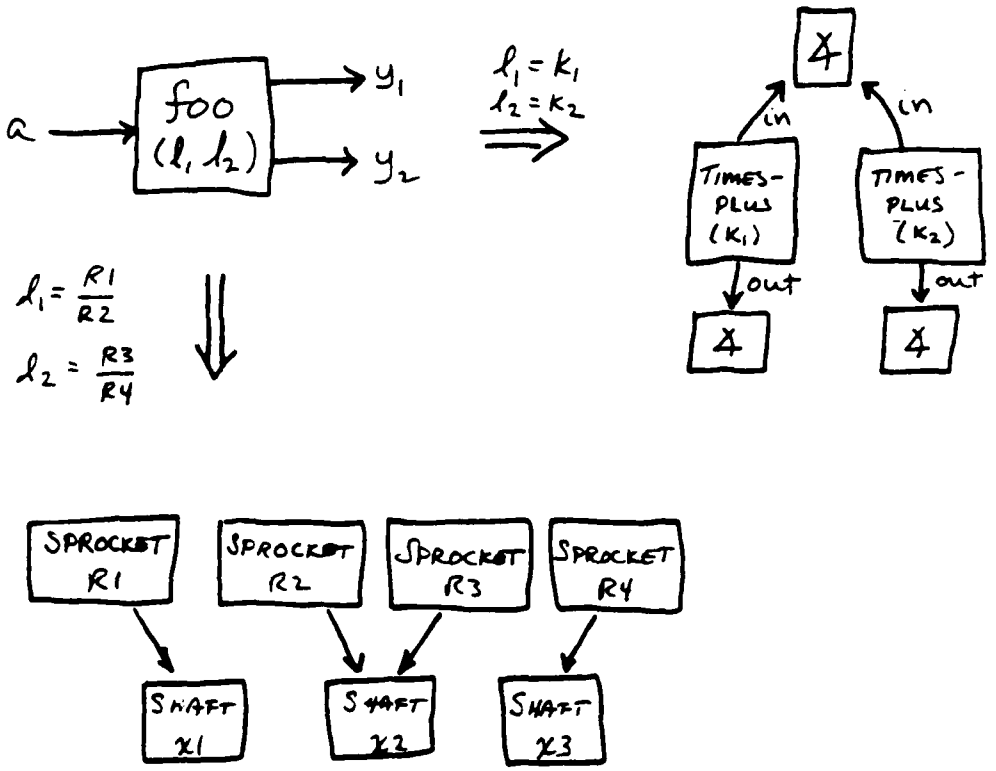


Figure 4.8: Parameterized Rules From A Learned Association: "FOO (l_1, l_2)" is the new non-terminal, one RHS involves sprockets, and the other is in terms of the TIMES-PLUS functional block.

demonstration purposes, but as this topic is covered at great length elsewhere[1][10], I will not go into it here. Suffice it to say that, as always, the crucial thing in creating generalizations is having the correct descriptive terms available. For discussion of the algorithm actually used by the system, see Section C.2 in the Appendix.

4.5 Summary

A precedent is defined to be a pair of different descriptions of the same behavior, together with variable correspondences. For convenience of implementation, I view one graph as being more high-level than the other.

In the current implementation of Precedent Analysis the algorithm attacks the problem of learning from examples by using the knowledge it already has to come up with a *maximal partial parse* of a precedent, then conjecturing a rule to finish the parse in a single step. The current implementation is merely the first pass at this task. Its main weaknesses are that it comes up with a single parse and forms a single rule (thereby possibly missing some conjectures), and it requires a large amount of search.

The algorithm can only fail when Criterion R also fails with regard to the partial parse. Beyond that, there is little to be said in general about when it fails. It can, however, be useful to learn rules that fail to preserve equivalence because they are useful in some common cases.

It handles parameters by grouping examples based on qualitative isomorphism and then calling a constructive generalizer.

Chapter 5

Learning by Failing to Explain: Rule Re-analysis

The last chapter dealt with how the system finds what is new about a single precedent. This chapter deals with how the system restructures the Grammar after it discovers a new, primitive rule. *Rule Re-analysis* is a method for making good use of a given set of precedents. Unlike the rest of the thesis, *the examples in this Chapter will assume that the system has no rules to begin with.* This is only done for clarity; the method applies equally to the general case.

Precedent Analysis may produce rules which are not very general, because there might be more than one unknown rule used in constructing the precedent. Thus, the learned rules will have RHSs which are some combination of more than one unknown rule. It is much less likely to see again a complex group of rule instances than it is to see instances of the rules singly. It is possible, however, later to learn new rules which allow one to tease apart the more general rules of which the first one was constructed. This leads to the idea of re-analyzing old learned rules in terms of newer rules.

For illustration, suppose the system learns the following two RHSs for the (made up) "NANDNOR" functional block:

$$y = (\text{OR } (\text{NOT } (\text{AND } a\ b))\ (\text{AND } (\text{NOT } c)\ (\text{NOT } d)))$$

$$y = (\text{OR } (\text{OR } (\text{NOT } a)\ (\text{NOT } b))\ (\text{NOT } (\text{OR } c\ d)))$$

Note that this implies it does not know either the NAND or NOR implementations involved. However, from a later precedent, it learns that the "NAND" block has the following two implementations: $y = (\text{NOT } (\text{AND } x\ y))$; and $y = (\text{OR } (\text{NOT } x)\ (\text{NOT } y))$.

If the system now goes back to the first pair of rules it learned, the NANDNOR rules, it finds that it *can* partially analyze them. For example, it can derive that

$$y = (\text{OR } (\text{OR } (\text{NOT } a)\ (\text{NOT } b))\ (\text{AND } (\text{NOT } c)\ (\text{NOT } d)))$$

is equivalent to the original:

$$y = (\text{OR } (\text{OR } (\text{NOT } a)\ (\text{NOT } b))\ (\text{NOT } (\text{OR } c\ d)))$$

Constructing the partial match between these expressions and eliminating the common parts, the system is left with the following equivalence: y = (AND (NOT c) (NOT d)) is equivalent to y = (NOT (OR c d)). This can get turned into two new rules for a new block, which corresponds to the NOR function.

Suppose the rules are always presented to the learning system in the best possible order. Might it not be, then, that Rule Re-analysis is a waste of time? The answer to this is no. This is demonstrated, by counterexample, as follows. Suppose that, unknown as yet to the system, there are four general design rules involved in constructing three precedents. The four design rules are as follows.

- $f_1(x) \Longrightarrow g_1(x)$
- $f_2(x) \Longrightarrow g_2(x)$
- $f_3(x, y) \Longrightarrow g_3(x, y)$
- $f_4(x) \Longrightarrow g_4(x)$

The three precedents are the following:

1. $g_3(f_1(x), f_2(y)) \equiv f_3(g_1(x), g_2(y))$
2. $f_2(f_4(t)) \equiv g_2(g_4(t))$
3. $[g_3(f_1(z), f_2(z)), z = f_4(w)] \equiv [f_3(g_1(z'), g_2(z')), z' = g_4(w)]$

Suppose that the Learning by Failing to Explain system is presented with these precedents in the order 1, 2, 3. On seeing 1, the system is not able to analyze it at all. Likewise, on seeing 2, the system can not analyze it at all. Thus far, the system has 4 rules: two rules implementing blocks representing each of the overall functions of the precedents[1].

Now, on seeing precedent 3, the system may analyze it using rules derived from precedent 1. This results in one new rule: $f_4(x) \Longrightarrow g_4(x)$. Rule Re-analysis applies this new rule to precedent 2. This results in the rule, $f_2(x) \Longrightarrow g_2(x)$. The system may then re-analyze the precedent-1 rules and arrive at two simpler rules. One has RHS $g_3(f_1(x))$, the other has RHS $f_3(g_1(x))$. Hence, the system is left with the following rules.

- $f_4(x) \Longrightarrow g_4(x)$,
- $f_2(x) \Longrightarrow g_2(x)$,
- $h(z, w) \Longrightarrow g_3(f_1(z), w)$,
- $h(z, w) \Longrightarrow f_3(g_1(z), w)$.

[1]...one rule for each graph of each precedent.

On the other hand, if one picks any of the six possible orders of presentation and applies Precedent Analysis without Rule Re-analysis, the set of rules conjectured is less general than the four rules. For example, suppose they are given in the order 1, 2, 3. Without Rule Re-analysis, Precedent Analysis conjectures the following set of rules as an addition to those made from each entire precedent. (h is a block created by the system.)

- $f_4(x) \Longrightarrow g_4(x)$
- $h(z,w) \Longrightarrow g_3(f_1(z),w)$
- $h(z,w) \Longrightarrow f_3(g_1(z),w)$

It thus failed to find the f_2 rule.

We decide which rule set is more general by asking which is capable of generating the other. Clearly, the set produced using Rule Re-analysis suffices to generate all the rules in the other set. However, there is no derivation of the f_2 rule in terms of the rules produced without Rule Re-analysis. Thus, Rule Re-analysis resulted in more general rules.

5.1 Summary

The algorithm presented gives an evolutionary view of design performance. As the system sees more and more precedents, it reorganizes its database by finding smaller and more general rules, thereby being able to derive more designs. The system's problem solving competence increases.

Two approaches to handling sequences of precedents are given: one which just accepts the precedents in order and applies Precedent Analysis to each, and Rule Re-analysis, which uses newly derived primitive rules to analyze old primitive rules. One uses less time per precedent, but is not as powerful as the other. The fact is that without re-analysis, the system requires *more* precedents to reach a given level of generality. Since precedents are in general much harder to come by than the time needed for Rule Re-analysis, it is clear that re-analysis is worthwhile.

Chapter 6

Conclusions

This report has been about a particular computational mechanism for generating conjectures as to structure/function design knowledge, and showing that the conjectures generated are useful in that they enable interesting design competences. A system has been implemented which illustrates these learning techniques. In [5], I document a prototype system which demonstrates the competences which are enabled by having knowledge encoded in a Design Grammar.

There is a question raised, however, by any work like this: under what circumstances is the given technique preferable to other techniques for solving the same task (if at all)? Although I do not have a complete answer to this question, I will approach it by comparing this technique to another method of acquiring design knowledge: Explanation-Based Generalization.

I will also discuss briefly a few general limitations and the relation to other work.

As usual, I have suggestions for future investigation which arise from this work. I have included these at the end of this Chapter. Appendix C deals with the related topic of the idea-history of the research. It, too, indicates potential future work: those areas I started but never finished for one reason or another.

6.1 Recapitulation

The research set out to investigate ways in which current knowledge could help constrain the search for new knowledge. As opposed to Explanation-based Generalization, however, it did not assume the ability to explain all input examples. The domain for learning is Design Knowledge. The problem of Design is too hard, however, to tackle all at once, so I separated the problem of learning about structure-to-function relationships from other issues, like controlling the search through the space of alternatives and taking into account other design specifications like cost and performance. This was partially justified by reference to the literature and was also partly an act of faith.

In learning research, one needs to state what it is that the system is going to learn, so that progress can be judged. To do this, I gave four design competences, which a system that does design should be able to exhibit. These were *Top-Down*

Design, Optimization, Analysis, and Analogical Design.

The thesis was that having knowledge encoded in a *Design Grammar* enabled these competences, and the Learning method given in Chapters 4 and 5 was a viable method of generating plausible conjectures about Design Grammar rules.

The Learning method presented here has two parts. The first is called *Precedent Analysis.* Its key idea is that in order to generate useful, powerful conjectures, one must *partially analyze the precedent to understand what is old, before attempting to formulate a rule about what is new.* This can be thought of as *Role Analogy*, because it finds devices that fill analogous roles in the two graphs of the precedent and conjectures rules based on the analogy. Criterion R provides some insight into when the conjectured rule will be true independent of the circuit context.

A kind of *rule base evolution* is a natural outgrowth of this technique. This is the second part of the Learning method: *Rule Re-analysis.* I gave an incremental algorithm that re-analyzes old rules using new rules, thereby finding even more general rules by cleaving the old rules along rule boundaries. This is inherently more powerful than not re-analyzing, even if optimal precedent ordering is assumed.

6.2 Explanation-Based Generalization

Explanation-Based Generalization[1] is a method of finding justifiable generalizations of single examples by explaining why the example satisfies the goal concept and then generalizing the explanation in an explanation-preserving way. The system can, in principle, build up a description of the precise class of instances to which the same general explanation can be applied to show concept membership.

The current efforts in the literature typically do not find the precise class; rather, they only find a subset of the class by finding a *sufficient* description which is not *necessary* for concept membership.

Other efforts have focused on refining a knowledge base by seeing where a faulty explanation breaks down.

6.2.1 The EBG Method

For a quick example of the technique, reconsider the example in Figures 4.5 – 4.6. The Learning by Failing to Explain learner conjectured the equivalence shown in Figure 4.6. An Explanation-Based Generalizer, by contrast, would be *given* the equivalence in Figure 4.6. It would also get an explanation of why the equivalence was true in the context of Figure 4.5. The explanation would be something like[2]

- On odd numbered clock cycles, both behaviors (in Figure 4.5) put out the value of x, because of the arrangement of the NOT-Z circuitry on the select line of the output MUX. Therefore, on odd cycles, from the standpoint of circuit outputs, the two subgraphs (Figure 4.6) are equivalent.

[1] Here are some references: [9], [12], [14], [17], [13].

[2] This is somewhat idealized. The actual details would be more complicated.

- On an even clock cycle, the output of the HLG is selected to be the output of the upper f box, whose output is $f(Z^{-1}(f(Z^{-1}(a))))$. Also, because of the NOT-Z circuitry on the output of the LLG, the y output is equal to the value at the output of the Z^{-1} box on even clock cycles. But because of the coordinated way the NOT-Z circuitry is set up on the input end of the LLG, the Z^{-1} box's output is also $f(Z^{-1}(f(Z^{-1}(a))))$, for even clock cycles. Therefore, for even clock cycles, the y outputs must be equivalent.

- Since every output is on either an even or an odd clock cycle, by these two steps the outputs are equivalent for all time. QED.

Looking at this explanation, the algorithm asks: what is it about the *context* which was crucial to this proof? From the third step, it is clear that it is sufficient that the y outputs by equivalent on both odd and even clock cycles. The even-cycle case was proved using the context fact that the y outputs were equivalent to the same functions of the outputs (in this case MUX) of the two subgraphs. The odd-cycle case depended on the fact that the outputs were equivalent functions of equivalent arguments (not necessarily MUX of a free input).

In some better formalism than English, it would be clearer exactly what the proof depended on, but one can imagine that the inference rules used were more general than the specific case shown. Intuitively, it is clearly overly specific to require that the y output be produced by a single multiplexor whose select line is tied to a one-bit counter and which has one input tied to a free variable, etc. As far as the proof is concerned, the only important characteristics were that on odd cycles, the outputs were explainably equivalent functions, and on even clock cycles they were explainably equivalent to the outputs of the subgraphs.

The Explanation-Based method would produce a description something like that in Figure 6.1 of the context required to make the explanation of equivalence hold. h_1 and h_2 stand for arbitrary functions, which can have any numbers of other inputs. Note this is now potentially more useful than the conjectured rule (from Figure 4.6), because it will always be allowable if it matches the situation.

6.2.2 Comparison

Learning by Failing to Explain is not an Explanation-Based Learner in the sense of the above definition. In fact, it only learns when it *can not* explain the situation.

There is no direct comparison between the two methods; they complement each other. It does not make sense to ask which is more powerful in general, as neither has its applicability domain included in that of the other. The point is that different situations call for different methods, and where one is most useful the other one won't be, necessarily.

Forming explanations for things is an inherently hard problem; after all, theorem proving is a special case of it. Any method which relies on being able to come up with an explanation every time it wants to generalize must fail to generalize sometimes. There are two reasons such a system could fail to generalize: its domain theory

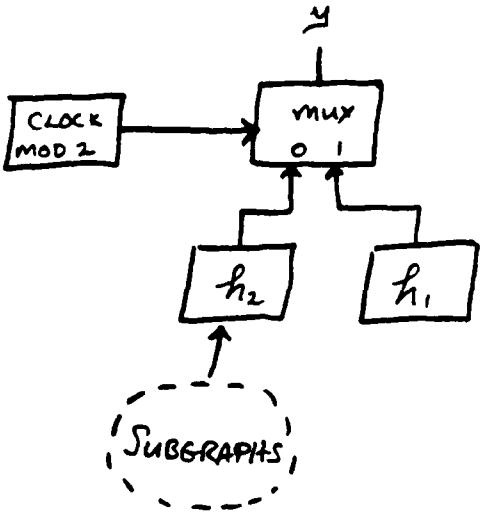


Figure 6.1: EBG-Produced Description of the Required Context.

is incomplete, so that no explanation exists, or the explanation is too hard for its explainer (parser, theorem prover) to find.

It should now be clear why a system which could only learn analytically would be limited: even if its domain theory were perfect, *i.e.* every true thing were in principle explainable, there would still be complex situations that it couldn't justify. This is not just mathematical nitpicking: the complexity of VLSI circuits provides a rich source of highly complex objects to explain.

A system which learns by explaining as much as possible and then making a reasonable conjecture, can deal with these complex situations in a useful way.

The other case in which Explanation-Based Generalization could fail is where the system's domain theory is insufficient to explain the situation, even in principle. Note, by the way, that this situation obtains in nearly all physical world domains to some degree, as there are always some special cases which aren't explainable in terms of the current theory[3] Not only does an exclusively explanation-based system fail to generalize a particular instance, it does not even have a hope of ever being able to generalize it. The most it could do would be to pile up a lot of highly specific instances which would probably never again be useful.

Contrast this with "inductive, syntactic" methods, in particular the Learning Algorithm given here: cases of in-principle unexplainable things are handled in the same way as in-practice unexplainable things: they are generalized as much as possible by partial analysis and then stored. Later, after the rule base evolves some more, the system could even go back and generalize further by discovering a rule which cleaves the original, mysterious example.

A syntactic learner could start without any knowledge of a domain and eventu-

[3]For example, the perturbations in the orbit of Mercury weren't explainable in terms of Newtonian Physics; the theory of Relativity was needed. Similarly, the classical theory of Relativity fails on the quantum level, etc.

ally hope to become an expert; an explanation-based learner could not.

On the pro side, however, there are clear situations where Explanation-Based methods are superior to inductive methods. A Physics student, on seeing that a bowling ball and an eight-ball dropped simultaneously off of a building land simultaneously, should not immediately generalize to "All black things fall at the same rate." [4] The student should use Physics knowledge to reason about gravity and air friction. In particular, this would avoid the *non sequitur* about the color black.

To summarize, both Learning by Failing to Explain and Explanation-Based generalization methods have places in a learning system. Use Explanation-Based on the cases which are obvious, and use Learning by Failing to Explain when mysterious situations arise.

6.3 Relation to Other Work.

It would seem that there is an interesting relationship between this work and that of Berwick [2]. Berwick's model of learning can be construed as a Learning by Failing to Explain method. His domain was natural language learning, where the grammars are, of course, string grammars. His mechanism attempted to parse an input sentence according to its current rules as much as possible, then *if the result satisfied certain criteria* the system proposed a new rule. His system did not attempt Rule Re-analysis. This is because (he argues) natural languages satisfy certain constraints which enable them to be learned in this manner. Thus, his system could be described as Precedent Analysis, together with some additional criteria regarding when to actually form a new rule.

Inasmuch as there is no reason to believe that the world of design obeys such a learnability constraint, it is not to be expected that Berwick's mechanism would work in learning Design Grammars from any kind of realistic examples. (Of course, if the most general rules were simply handed the system as precedents, any system could learn that way.) It is possible, however, that the use of Rule Re-analysis can substitute, at least in part, for the missing learnability constraint. It is also possible that more powerful language acquisition could occur if Berwick's method incorporated Rule Re-analysis. Of course, fidelity to psychological data might not be maintained.

6.4 Limitations and Suggested Future Work

Experimentation with some examples on the system (most notably those documented in Appendix E) reveals the following limitations of both the basic idea and the current implementation approach.

[4] This would be wrong: a bowling ball and a black piece of paper do not fall at the same rate off of a building.

- Sometimes, the *maximal* partial parse is not the most desirable partial parse to use. In some cases a much more context independent rule can be obtained if a non-maximal partial parse is used.

- This method only finds *one* partial parse. In some cases it may be desirable to find many candidates, giving potentially different rules.

- The greedy algorithm can be too greedy, causing the system to miss a better partial parse.

- The system needs some better approaches to search control in the analysis algorithm. In particular, some method of focusing attention on sections of large graphs would be much more efficient. Adding a simple chunking mechanism for learning useful derived rules would help greatly.

These of course suggest that one major area of future research is investigation of control strategies and knowledge for guiding the Analysis process.

In addition, here are some higher level research questions raised, but not answered in this work.

- As emphasized throughout the thesis, I did not address how search control knowledge could be learned and used by the system. In particular, there is a need for the study of specifying constraints other than function, like cost, performance, and resources, and using these to help guide the search. There has been some work on learning search heuristics from problem-solving traces, for example Mitchell[11]. Much more thinking remains to be done to learn these things in complex domains.

- More ways of creating useful descriptions of devices are needed. For example, it would be nice to collapse the rules for 2-bit, 3-bit, 4-bit,...,*n*-bit adders into a single parameterized rule without the necessity of the teacher providing it explicitly.

- It would also be useful to be able represent generalized functions by having a vocabulary of descriptive terms arranged in a generalization hierarchy. For example, the system learned in Chapter 4 that one can slide delay boxes (Z^{-1}) from output to inputs of an XOR box. It would be nice if the system could somehow generalize this to any "combinational" block, not just XOR.

- One of the original intentions of this work was to explore problem solving in general, not just in design. Does this work extend to other problem solving formalisms and domains? My intuition is that it does, because in most problem solving domains there seem to be (1) notions of structure: the domain operations and connections between their use; (2) notions of function: what goals the operations perform and how to combine goals to achieve more complex goals; and (3) problem-solving "techniques" (rules, schemata, etc) which associate structural constellations with specific goals. A solved problem constitutes a precedent, so it would seem that a learner could try to understand

as much of the precedent as possible, in terms of the techniques it already knows, before conjecturing new rules.

Here are some implementation oriented directions which are worth thinking about.

- The parser currently implemented could be improved a great deal through further research. In particular, it is essentially a top-down parser. A combination of top-down and bottom-up methods is probably better. It must, of course, still produce partial parses so the Learning Algorithm can still use it. Another idea is to use the Design search heuristics to speed up the top-down parsing, reasoning that any designed precedent was probably designed using similar heuristics to guide its search.

- The system currently looks through all its rules and tries (graph) matching each to the problem. It would be desirable to avoid some of the matching involved. This can be alleviated by a parallel architecture, as each precedent can be checked separately. Also, it is not necessary to rematch a rule to the entire design if only a small part has changed. If the graph matcher could be focusable to a neighborhood of the changes, significant speedup could be obtained.

- A marriage between the Learning method discussed here and the Explanation-Based methods discussed elsewhere[12] would seem to combine the best of both worlds: an inductive method which can learn even in situations it can not fully understand, and an analytic method which finds justifiable generalizations in well-understood situations. After all, it is silly to use superstitious (inductive) methods when you know an explanation. But on the other hand if you can't explain something (either because you can't in *principle* prove it, or because you can't in *practice* prove it) you still need some method that will produce plausible conjectures.

- It would be interesting to apply the method to other design domains, like programming.

Appendix A

Graphs, Grammars, and the Parsing Problem

A.1 Graph Grammars

A Design Grammar is a slightly restricted form of a *graph grammar*, a formal entity, whose definition is the subject of this section.

A.1.1 Graphs

A graph[1] *is a quadruple* (V, E, T_v, T_e) *where*

- V is a set of ordered pairs, each of whose first element is a primitive thing called a vertex and whose second element is an element of T_v. I will overload the term *vertex* by calling elements of V vertices as well as the first element of the ordered pair. I will also refer to a vertex as a *node*.
- E is a set of ordered triples. The first two elements of each are vertices, and the third element is an element of T_e. Each of these shall be called an *edge*, *arc*, or *link*, interchangeably.
- T_v is a set of primitive things called *node types*.
- T_e is a set of primitive things called *arc types*.

If T_v and T_e are singletons, the definition coincides with the standard definition of directed graphs. If these conditions obtain and if for every edge $(x, y, \cdot)$ in E the edge $(y, x, \cdot)$ is also in E, then we get the standard version of undirected graph[2].

Two graphs G_1, G_2 are *isomorphic* if and only if there exists a 1-1 correspondence f that maps V_1 onto V_2 such that $A \quad (x, y, t) \cdot \quad E_1$ if and only if

[1] This definition is more general than the usual definition of a graph one sees in graph theory. It subsumes both the notion of "graph" and that of "directed graph" as special cases.

[2] Well almost. This kind will have twice as many things called edges as a standard graph would, but this is a mere technicality.

$fA = (f(x), f(y), t) \in E_2$, and so that if $v \in G_1, f(v) \in G_2$ then the type components of v and $f(v)$ are the same.

A graph $H = (V_H, E_H, T_{vH}, T_{eH})$ is a *subgraph* of a a graph $G = (V_G, E_G, T_{vG}, T_{eG})$ if and only if

- $V_H \subseteq V_G$, and
- For every $v_1, v_2 \in V_H$ and for every $t \in T_{eG}, (v_1, v_2, t) \in E_H$ if and only if $(v_1, v_2, t) \in E_G$

A node, v is a *connection point (cp)* of a subgraph H of a graph G if and only if there is an arc A between v and some node w that is not in the subgraph.

A.1.2 Grammars

A grammar is a formal scheme that encapsulates the rules governing a language. A language (in this case) is a particular set of graphs. A grammar for the language is a set of production rules that *generate* all and only the allowed graphs in the language. That is, a graph is in the language if and only if there exists a string of productions starting from some single non-terminal node (see below) and ending with the desired graph.

A *rule* (production rule) is an ordered triple consisting of a graph, a graph, and a mapping from connection points of the first to connection points of the second. Call these the LHS, RHS, and the mapping respectively.

A rule can be used by applying it to some graph G. That is, if the LHS is isomorphic to some subgraph H of G, and connection points are preserved under the isomorphism, then H can be removed from G and the RHS inserted in its place with the mapping dictating how the connections are made.

If the LHS consists of one node which is not a connection point and all the rest connection points, then term that graph a *non-terminal graph*.

A graph is in the language generated by a graph grammar if and only if there exists a finite sequence of rules such that the first rule's LHS is a non-terminal graph and the result of applying the rules in order is a graph isomorphic to the graph in question.

The *recognition problem* for a grammar is to determine for an arbitrary graph whether it is generated by the grammar. The *parsing problem* for a graph and a grammar is to determine, given that the graph is generated by the grammar, a sequence of productions that produces the graph.

A.2 The Parsing Problem

The purpose of this section is to prove that the recognition problem for Design Grammars is uncomputable. In fact, I'll show that every recursively enumerable language is the language of some Design Grammar. Thus, because the set of Turing Machines that halt on their own numbers is r.e., and not recursive, there can be

no algorithm that can take a graph and a Design Grammar as inputs and decide whether the Design Grammar generates the graph.

I'll also remark that every Design Grammar generates an r.e. language.

Theorem. *For every recursively enumerable language L, there exists a Design Grammar that generates it.*

pf. The form of the proof is a reduction. It is known that every r.e. language is generated by some *Type 0 (string) grammar.* I'll show that by encoding strings as linear graphs, a Design Grammar can be constructed that generates exactly the encodings of the language's strings.

A Type 0 Grammar is a finite set of (string) productions of the form $\alpha \longrightarrow \beta$, where α and β are strings over the finite set of symbols. The symbols may be terminal symbols or non-terminal symbols. There is a distinguished non-terminal symbol, S, called the start symbol. The only restriction on the rules is that α may not be ϵ, the empty string. β may be. A string of terminals is generated by the Type 0 grammar iff there exists a finite sequence of rule applications starting from the single symbol S, which ends with the desired string. A rule application is the act of replacing the LHS as a substring of the current string with the RHS.

Lemma. *Every r.e. language is generated by some Type 0 Grammar.*

pf of Lemma. The reader is referred to Section 9.2 of *Hopcroft and Ullman*[8] for a real proof of this. The basic idea is that Type 0 Grammars are so expressive that one can encode any Turing Machine's finite state transitions as grammar rules, with the current string acting as the TM's tape. The input is the string asked about, and the grammar rules act on the string, mimicking the action of the TM on the string. If the start symbol is ever reached, then the grammar "accepts" the string, otherwise not. Hence, deciding whether a given string has a grammar derivation is at least as hard as deciding when some TM accepts a given input. But because one could do this for any TM, and because a language is r.e. if and only if there exists some TM which accepts it, one concludes that any r.e. language is generated by some Type 0 Grammar. **QED Lemma.**

Now, given a Type 0 Grammar, G, construct a Design Grammar, D as follows. Define node types, one for each distinct symbol of the vocabulary of G. In addition, define a node type, distinct from all others, called CP (for Connection Point). Also define a single arctype, r, interpreted as "to the right".

To encode a string w as a graph, create $|w| + 1$ nodes of type CP and one node of the appropriate type for each symbol of the string. Starting and ending with connection points, build a graph using the nodes by connecting them in a linear sequence with the arctype r, so that every odd numbered place is a connection point and every even numbered spot is a string symbol node. Preserve "to the right" order with the r links.

Now for every rule i of G, create a distinct non-terminal symbol, denoted R_i. Build two rules of D: each has as LHS an encoding of the string "R_i". One has as

RHS the encoding of α, and the other has as RHS the encoding of β. Record $R_i \longrightarrow \alpha$ as equivalence-preserving, and record $R_i \longrightarrow \beta$ as non-equivalence-preserving.

Now by the rules of use for Design Grammars, a terminal string has a derivation in G if and only if the graph that encodes it has a derivation in D. This is because the extra non-terminals created appear on exactly one side of two rules, so if they ever appear in the current graph, they must be eliminated by one of the two rules. If one is eliminated by the same rule that introduced it, then it is as if the two had never happened at all. Alternatively, if one is eliminated by the other rule, then it is exactly as if the corresponding grammar rule had operated on a section of the string. Note that this is true even if other rule applications intervene between the introduction of the special non-terminal and its elimination: the other rules can't affect the non-terminal because they can't match it. Also, the operation of the graph rules only allow replacing α by β, not β by α, because the "β rule" is non-equivalence-preserving.

Now suppose that a D derivation exists, starting from the non-terminal graph encoding "S". By reordering the operations, we can put the introduction and elimination of any special non-terminal together in the sequence. Then we simply transcribe in a straight-forward way the corresponding G derivation.

Conversely, given a G derivation, we may even more straight-forwardly write down a D derivation by substituting pairs of graph rule applications for single rule applications. **QED.**

Theorem. *Suppose D is a Design Grammar with non-parameterized graph elements. Given any starting graph in the language of D, D generates an r.e. language of graphs from it.*

pf. The only real technicality here is that a Design Grammar, as defined, has no distinguished start symbol. This is taken care of by allowing any graph to be the start symbol and taking the particular one as input.

A very straight-forward breadth-first, British Museum algorithm suffices. Start with the input graph on a queue. Do until the queue is empty (which may be never): Consider the graph on the head of the queue. If it consists of terminals only, and if it hasn't been output before, then write it out as an output. Otherwise, for every one of the finitely many possible rule applications, generate the result of applying it. Put these on the back of the queue. Repeat.

Clearly, if some graph has a derivation in D, then the algorithm will eventually put it out. If not, then it won't, because the algorithm puts out only graphs which appear as the result of a sequence of applications of rules. **QED.**

Note that if the rules are parameterized, then they may be parameterized by real numbers. Because there are potentially uncountably many terminal graphs in that case, no TM can exist that enumerates the them. Thus, the theorem won't be true of arbitrarily parameterized Design Grammars.

Appendix B

Matching via Constraint Propagation

The purpose of this Appendix is to explain the subgraph isomorphism algorithm used by the implemented system. The task of the algorithm is to take in a large graph and a small graph and put out a list of all possible matches (subgraph isomorphisms) between the two.

The intuition behind the algorithm is that for almost any pair of graphs there are a large number of ways of trying to match them which are obviously ridiculous. For example, a candidate match might try to associate a node of one type to a node of a different type. Alternatively, the types might match, but the node of the small graph has more neighbors via a given arc type than does the associated node.

If an algorithm could eliminate these poor guesses at the outset in a small amount of time, then the system could eliminate some search. As remarked previously, if there are actually many matches (isomorphisms) between the graphs then it takes a long time to find all of them simply because there are many matches. Any algorithm must take a long time on those cases.

The interesting case (the situation which prevails most often, at least in this system) is when the small graph is not a subgraph of the LG at all. One would like an algorithm to find this out quickly if possible. From the limited set of cases which the system tried this on, graphs used by the design and learning systems, the empirical results are encouraging: it seems that most such problems have lots of obviously bad possible matches. The algorithm behaves very well on them.

B.1 The Algorithm

A match is represented by a data structure I'll call a graph-match (GM). It is essentially an association list: for each node of the small graph there is a length one list of nodes (possible-match-set or PMS) of the other graph to which it may be matched. I will term the domain of the GM, *i.e.* the "keys" of the alist, the left-set or simply the domain. I will term the range of the GM, the union of all PMSs, the right-set or simply the range.

More generally, a GM may associate more than one right-set node to each left-set node; in that case it does not represent an isomorphism of graphs, just an association of nodes.

Initialize a GM with left-set equal to the set of nodes in the small graph and each possible-match-set to the set of all large graph nodes whose type and parameters are compatible with the given left-set node. (Intuitively, the PMS of a left-set node contains nodes which remain possibilities for matching to the left-set node in an isomorphism. The algorithm will see to it that no node is eliminated from a PMS if the association can take part in an isomorphism.)

Call Algorithm SG with the initial GM.

ALGORITHM SG

- Call subroutine P with the initial GM and a queue of all nodes in the left-set. Subroutine P is defined as follows, taking a GM and a queue of left-set nodes to check. It puts out an altered GM, one with all ridiculous possible node matches eliminated.

  SUBROUTINE P

  - While the queue is not empty, for the head of the queue, n, and for each possible match, m, in its PMS, check to see if constraint LC (definition below) is satisfied by (n, m, <current GM>). If so, do nothing; if not, eliminate m from from the PMS of n.
  - When all possible matches have been tried for n, if any were eliminated, then adjoin all neighbors of n to the back of the queue (if they are not already on it), and process the next queue entry.
  - When the queue finally empties (as it must since it only keeps going as long as nodes get eliminated from the GM) P checks constraint GC (defined below) and eliminates matches as necessary from the GM. If constraint GC eliminates any possible matches, make a new queue from all neighbors of those left-set nodes which had matches eliminated and go back two paces and process the queue again.
  - By getting to this point, both LC and GC must pass the GM without eliminations, so P returns the latest GM.

- If the returned GM has all PMSs empty, then halt, returning the empty list.
- If the returned GM has all PMSs singletons, then halt, returning the length one list consisting of the returned GM.
- Otherwise, call routine SEARCH, passing the current GM.
- Halt, returning SEARCH's result.

Routine SEARCH is defined as follows, taking a GM as input and returning a list of isomorphisms.

ALGORITHM SEARCH

- Establish a list (initially empty) to hold the isomorphisms to be found.
- Pick some left-set node, n, whose PMS is not a singleton. (One such will exist, by construction.) Save the input GM somewhere.
- For each element, m, of n's PMS, do the following.
  - Build a new GM identical to the original, saved version, except put n's PMS equal to the singleton containing m.
  - Call SG with the new GM and a list of all the left-set nodes in the new GM. Append its results to the list of results.
- Halt, returning the list of results built up by the loop.

Note that by definition of isomorphism it must be that if a node is *not* an a-neighbor of n then it can match no a-neighbor of m, else m is an invalid match. Thus, for each arc type a, we define a negative arc type $< a >$, and enforce that every pair of nodes n_1, n_2 of either graph are either a-neighbors or $< a >$-neighbors[1].

CONSTRAINT LC is a local constraint which must be satisfied by a candidate node match and the current GM (n, m, gm): For any arctype a there must be a one-to-one[2] mapping from the a-neighbors of n (neighbors of n via arctype a) to the a-neighbors of m such that each image node of a neighbor is in the PMS of the neighbor in gm.

CONSTRAINT GC is a global constraint on the entire GM. Essentially, there must exist a one-to-one function mapping left-set nodes into right-set nodes such that the image of a left-set node lies in its PMS.

GC finds sets of left-set nodes which are "critical." A set $\{n_1, n_2, ...n_p\}$ of left-set nodes is critical if the union of their PMSs has at most p nodes. If it has fewer than p, then no global isomorphism could exist, so GC sets all PMSs of GM to the empty set, indicating failure. If the critical set has exactly p nodes, then no other node of the left-set may have a PMS which contains a range node of the critical set. Thus, GC returns a GM with the appropriate possible matches eliminated. It does this for all critical sets in GM before returning the result.

Both of LC and GC may be implemented in a straight-forward way using the polynomial-time algorithm for finding a *maximal graph-matching* in a bipartite graph[3], as GMs may be viewed as bipartite graphs.

[1] This *does not* force the graphs always to have the extra baggage of the negative arc types explicitly represented. It is quite simple to decide which possibility holds just from x being present or not.

[2] f is one-to-one if and only if $f(x) \neq f(y) \Rightarrow x \neq y$.

[3] This is something of a pun, since the usual (graph theory) meaning of graph-matching has nothing to do with matching two different graphs. For definitions and the polynomial-time algorithm see for example Sedgewick 16.

B.2 Correctness

Theorem. *The GM g is a subgraph isomorphism of graph s to graph l if and only if g is in the list put out by Algorithm SG on input i, where i is a GM associating each node of s to the set of all nodes of l which are type/parameter-compatible to it.*

pf. ($\Leftarrow$) Suppose g is in the list produced by Algorithm SG applied to the appropriate i. By construction, SEARCH only returns results produced by some call to SG. SG only returns a match if it is all singletons. Thus g must be all singletons. Thus, every node of s has exactly one associate in g. g must be one-to-one; otherwise there would be two or more nodes of s which form a critical set whose range is of size one. Thus GC would have disallowed g.

Denote the unique element of the PMS of n by $g(n)$.

There are now two cases in which g could fail to be an isomorphism. (1) There are two nodes n_1, n_2 of s such that n_1 is an a-neighbor of n_2, but $g(n_1)$ is not an a-neighbor of $g(n_2)$. This can't be true, for LC must have checked each of the possible matches at least once (we called SG initially with the queue consisting of all nodes) and the *last* time LC checked n_1's PMS, LC would have failed, because by then all neighbors of n_1 would have had singleton PMSs (else when they became so, n_1 would have been checked again), and the only possible map of the a-neighbors of $n1$ compatible with g would not have associated n_2 to an a-neighbor of $g(n_1)$. Thus (1) can not hold for any arc type.

(2) There are two nodes n_1, n_2 of s such that n_1 is not an a-neighbor of n_2, but $g(n_1)$ is an a-neighbor of $g(n_2)$. This must also fail to hold, because if it did then case (1) would hold for the arc type $< a >$. But case (1) can't hold for any arc type. Thus, (2) can not hold either.

Therefore, g must be an isomorphism of s to l.

($\Rightarrow$) Suppose g is an isomorphism of s to l. If f is any GM whose left-set is the nodes of s, I will say that g is compatible with f if and only if $g(n) \in \mathrm{PMS}(n)$ under f. By definition of isomorphism, g must respect node types/parameters, so g is compatible with i.

I will first show that P never renders g incompatible with the current GM through elimination of a match. For P to do so, either LC would have to fail at a node n and match $g(n)$ in current GM h, or GC would have to eliminate $g(n)$ from PMS (n) under h.

Suppose g is compatible with h and LC considers the match $n \leftrightarrow g(n)$ in h. Clearly, if $a(n)$ is the set of a-neighbors of n, the one-to-one mapping $g\,|_{a(n)}$ is compatible with h, so LC won't eliminate $g(n)$ from PMS(n).

Suppose g is compatible with h and GC considers h. Furthermore, suppose it finds a critical set, $\{n_1, ..., n_p\}$. Since for each n_j, $g(n_j) \in \mathrm{PMS}(n_j)$, the h-image of the critical set must be exactly the g-image. Then suppose that some other node, n has a match in the h-image of the critical set. Then GC will eliminate that association. But that association can not be $g(n)$, else g would not be one-to-one, by the Pigeon Hole Principle. Thus, GC will not eliminate any g associations.

Thus, since neither LC nor GC can eliminate a g-association, P can't either.

Thus, if SG halts without calling SEARCH, g will be returned by SG.

Suppose that whenever SG is called with a GM h, with which the subgraph isomorphism g is compatible, and whenever h forces SG to at most n levels of recursion in calling SEARCH, SG returns g among its outputs. Also suppose that the GM i, with which g is compatible, forces SG to recur to exactly $n+1$ levels. Since g is compatible with i, the first call to P will not eliminate any g-associations from the current GM, so SEARCH will be called with a g-compatible GM, i'.

SEARCH will pick some node, q, and successively call SG with GMs i'' which have PMS(q) set to a singleton from the i' PMS. In particular, then, SG will at some point be called with PMS(q) in i'' being the singleton containing $g(q)$. But in that case, as the algorithm is already at a depth of one in recursion, i'' can only force SG to n levels. But notice that g is compatible with i''. Hence, by the induction hypothesis, the recursive call to SG will return g among its outputs. Then the top-level call to SEARCH will return the union of all of its calls to SG, so its output will include g. The output of the original call to SG is just the result of the call to SEARCH, so SG will return a list containing g.

Hence, by induction on n, Algorithm SG will always return g whenever it is an isomorphism of s to l.

Combining the two halves of the proof, it is now clear that Algorithm SG finds all and only the isomorphisms from s to l. **QED.**

B.3 A Few Words About Complexity

The first remark is that Subroutine P must run in time polynomial in the size of the graphs involved. LC and GC may only be called as long as the current GM has non-empty PMSs. Each constraint only serves to decrease the size of the PMSs. At most mn match-eliminations, where the small graph has m nodes and the large graph has n nodes, can occur, because this is the largest number of elements of PMSs there can be initially. Hence, at most $m + nm^2$ (each elimination can cause all m nodes to be added) queue entries can be made. Since GC is only called after LC's queue is emptied, then GC can be called at most (assuming LC's queue is emptied $2(m + nm^2)$ times) $2(m + nm^2) + 1$ times. As remarked above, LC and GC run in time polynomial in m and n, so let $p(m,n)$ denote the worst of the two[4]. Then the entire run time of Subroutine P is at worst $q(m,n) = (2(m + nm^2) + 1)p(m,n)$, which is polynomial in m and n. This is a very loose upper bound.

The second remark is that if SEARCH is called to a recursion depth of d, $0 \leq d \leq m$, then Algorithm SG requires less than or equal to $q(m,n)\sum_{i=0}^{d} n^i$ which is less than $q(m,n)n^{d+1}$.

The algorithm can be viewed as a standard search algorithm that is augmented by a polynomial-time "filter" which cuts out a large amount of the branching and

[4]The time complexity of the bipartite graph matching algorithm dominates their complexities. Sedgewick [6] quotes this at $O(V^3)$ for graphs with V vertices. In our case, we can identify V with $m + nm$ in the worst case, giving a loose upper bound of $(nm)^3$.

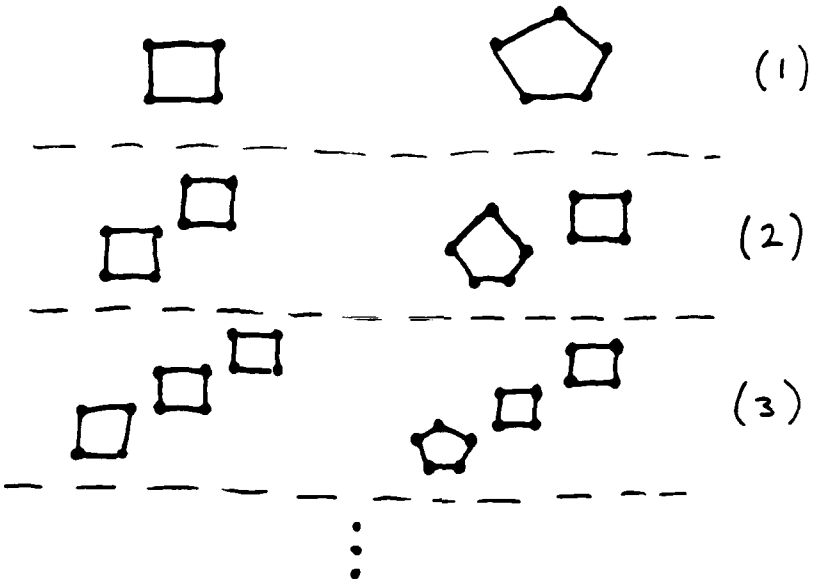


Figure B.1: An Exponential Class of Graph Pairs

depth of the search tree.

The question remains, under what circumstances does the algorithm behave poorly? The high-level answer is that it behaves poorly when the graphs are highly homogeneous: few arc types, few node types, lots of symmetry, highly similar node-degree distributions, etc.

For example, consider the class of graph pairs schematically depicted in Figure B.1. The nth pair is constructed as follows. The small graph consists of n disjoint squares and the large graph consists of $n-1$ disjoint squares and a disjoint pentagon. All nodes are of the same type and all arcs are duplicated so they go both ways. The algorithm goes to at least level n of recursion on the nth pair. This is exponential behavior.

B.4 Relation to Other Work

Ullman [20] gives a fast algorithm which is basically the same as Algorithm SG, but he does not employ anything like Constraint GC. He is also solving a slightly different isomorphism problem: he allows two nodes which are not neighbors in the small graph to be neighbors in the large graph. One could convert Algorithm SG so that it allowed this simply by ignoring negative arc types. He gives some empirical results, based on an assembly language coding of the program, which reflect the experience I've obtained. The algorithm behaves polynomially on randomly generated graphs; this is reflected in the good performance on all the graphs encountered by my system in practice. When a referee of his paper suggested he try the program on a class of strongly regular graphs, the performance went from 2 seconds for random graphs of size roughly 25 nodes, to greater than 3000 seconds on regular graphs with 25 nodes. This reflects the poor behavior of Algorithm SG on highly regular

graphs.

Haralick and Shapiro ([6],[7]) formalize the notion of the *Consistent Labeling Problem.* It is a generalization of numerous problems, such as Subgraph Isomorphism, Latin Squares, Graph and Automata Automorphism, Line Labeling, and many others. It consists basically of a set of units (things to be labeled), labels, and a constraint relation which restricts the allowed assignments of labels to units. They define a two parameter class of local constraint operators which essentially generalizes Constraint LC above. They do not seem to use the Global Counting technique of Constraint GC above. Their basic conclusion seems to be in line with Ullman's: in practice, the algorithm performs well on graphs derived from real life situations, but can perform poorly on regular (homogeneous) graphs. I believe that Constraint GC can be generalized to the general consistent labeling problem when the labeling must be one to one.

B.5 Summary

This Appendix has given an algorithm for solving the subgraph isomorphism problem on graphs with labeled nodes and labeled arcs. The reason it is of note is that it allows the matcher to take into account domain constraints to limit the search involved in this *NP*-complete problem. The domain constraints are encoded into the representation through the node- and arc-labels.

In fact the algorithm actually finds all subgraph isomorphisms between the two graphs, rather than just returning yes/no, so in the case where there are many matches between the two graphs the algorithm will take time at least proportional to the number of matches.

Experience with hundreds of "real life" graphs which came up in testing the learning/design system has shown that whenever there was no match between the input graphs, the algorithm went no deeper than one level of recursion, so behaved polynomially. This is in line with the empirical results of others who have used algorithms similar to this one. The actual implementation chosen here was a prototype version designed more for convenience in debugging than speed, so I expect that a tight coding could increase its speed quite significantly. (It won't change the depth of recursion, of course.)

Appendix C

Thesis History

The intent of this Appendix is to outline two of the ideas which did not make it into the thesis. One failed to make it because it turned out to be a bad idea; it is nevertheless an idea which might occur to someone else, so it is important to say why it did not work out. The other idea is mentioned because some aspect of it may be worth pursuing.

C.1 Representing Constraints Other Than Function

One of the initial topics of interest to me was analogical problem solving. Specifically, how might a program use previous problem-solving experience to constrain the search for a solution to a design problem? The interest was in applying analogy to all classes of constraints in a problem at once.

It is easy to think of scenarios of this type of behavior; for example, "I need a low cost mechanism to transfer low power at low speeds between shafts in this sewing machine. That dishwasher transfers low speeds at low power in its agitator, and uses a cheap belt drive mechanism. I'll try a belt drive in the sewing machine, by analogy with the dishwasher."

There is also the complementary use of analogy: "I need a low cost mechanism to transfer low speeds between shafts in this sewing machine. The use of jeweled bearings in the high speed drill caused it to cost ten thousand dollars. Ten thousand dollars is expensive for a drill. By analogy with the drill, I will *not* use jeweled bearings in the design, because even though it performs a necessary part of the function and satisfies the power and speed requirements it is likely to be too expensive."

There are many more such scenarios, where a designer makes a design decision based on knowledge of how it worked out in a similar situation. This is still an interesting sort of problem-solving behavior; I think it needs further investigation. A few questions arose, however, during the thesis work.

Specifically, this suggests the following problem-solving paradigm: start with an initial design specification, then pick the rule or precedent which matches the best to its constraint specifications, then try it out. If it is a precedent, then it may be

necessary to throw out certain things which are not relevant. This can be viewed as a form of GPS, in that the program looks at the constraints left to be satisfied in the goal description and picks the operator (precedent or rule) which seems to get the closest.

In this scheme, many different classes of constraints need to be represented, like function, resource usage, and performance. In a typical real-world gear problem one might need to represent all of the following: shaft-speed ratios (function), space usage, cost, power transfer, operating speed range, even design time of the device.

Suppose we have such a representation for each precedent, rule, or problem. How well would the proposed problem-solving method work? The key observation is that *similarity is a good heuristic for some classes of constraint, but it seems to fail miserably for others.* Specifically, it seems to work well for function constraints; that is what Analogical Design is about. But consider space usage constraints. It could be that a precedent problem was solved subject to identical space resource requirements as those imposed on a given problem, but because of differences in other constraint specs, the precedent's solution is bad in the problem context in that it leads to large cost, insufficient durability, or some other problem. Clearly our similarity heuristic would take into account the identical space requirements and weight that precedent higher.

Note that this is not a statement about the *domain* itself, rather about *classes of constraints* present in any design domain.

To make matters worse, suppose *all* of the problem's constraint classes have identical problem specs to some precedent, except one crucial specification. Also suppose that the crucial one acts to rule out the precedent's solution completely (*e.g.* make the same device, but at one tenth the cost). It would appear that the similarity between problem and precedent is overwhelming in this case, but to use it would be totally wrong.

The problem is that these types of constraints satisfy all or nothing matching criteria: a proposed solution is no good unless it completely satisfies the resource, cost, performance constraints. Contrast this with functional constraints. It *is* useful to try a proposed solution which only partially matches the functional constraints; some other partial solution can match the rest. An example of this would be combining a crankshaft, an idler gear, and a reduction gear to make up an entire mechanism. Each piece satisfied some aspect of the function, but not the entire specification. It is to no advantage to try a solution which *almost* stays within the spatial limits of the problem, or costs just a bit over the upper cost bound. On the other hand, if the candidate solution *does* satisfy the constraints, then it seems intuitively that one should try the one which uses the fewest resources; but that would be the one which was the worst match to the problem's resources.

One idea for getting around this problem would be to have some representation where each type of constraint had a description in some type of representation where partial matches did combine in the way function does. For example, if the spatial constraints of a gear problem were translated into the form, "first transfer power down this narrow channel, then around the obstacle, then across the long, open space," then partial solutions would combine. One part could transfer power down

the narrow channel, one around the obstacle, and one across the open space.

One problem with this scheme is that even though it might work for spatial (resource) constraints, other constraints do not seem to lend themselves to it. Consider cost. "First spend less than a thousand dollars, then spend less than another thousand dollars, then spend less than eight thousand dollars." If this representation is useful, where did it come from?

Another difficulty is that design problems are not usually stated in these terms. Usually, the problem is just one large, complex mass of interacting constraints. Even if the initial problem is of this nice form, the process of introducing structures as partial solutions changes the resource descriptions drastically, so the intermediate stages would have to be re-expressed. How this dynamic representation changing might take place is mysterious.

It was about at this point that I decided to concentrate on functional complexity, since those constraints seemed to be amenable to analogical problem solving in a GPS sort of framework.

C.2 Leveled Closure Approach to Generalization

As mentioned in Chapter 4, to deal with parameterized representations the system must do constructive generalization [1].

The approach I tried is the following very general one. I suppose the system has a store of knowledge in the form of representation-maintaining subroutines. That is, any representational predicate which the system knows has a subroutine which is capable of deciding whether it is true. For example, there might be such a subroutine which can tell whether its two integer inputs are equal. Also, there are "functions" which are little algorithms which take in values and return new values. There might, for example, be a binary function which returns the sum of its two integer arguments.

When the system sees two descriptions which are supposed to represent instances of the same concept, it tries every subroutine it knows on every combination of the parameter values (which are type compatible with the subroutine) to decide what the maximally specific description is that is true of both instances. When every possible predicate is found, we say the program has reached the *closure* of the current description. In fact, we'll call the closure of the two input instances the *level 0 closure.*

Suppose the system goes along seeing positive and negative instances, possibly throwing relation-instances out of the current generalization-description if the new positive instance does not satisfy them. If it ever ends up throwing away all of them, so that the current generalization-description becomes empty, then a crisis has occurred.

[1] *Constructive Generalization* is generalization where the program must not only climb a generality hierarchy, but it must create the generalized descriptions from lower level primitives as well. Having an *a priori* description language, in terms of which the concept is guaranteed to be expressed, is a powerful constraint.

The program takes the following action. Call all of the known functions on each type-compatible combination of values, and create a new set of derived values for each instance. For example, compute all sums of integers and denote each different sum by its symbolic representation (*e.g.* (SUM (GEAR-1 RADIUS) (GEAR-2 RADIUS))). Then compute the closure of the augmented descriptions. Call the result the *level 1 closure.*

Continue receiving examples until either the examples stop coming or another crisis occurs. Each time a crisis occurs, compute the next level's closure. Eventually the correct description will emerge, assuming it is representable.

This is very general, in that it does not require knowledge of the semantics of the representational predicates. Also, the number of new representational entities examined is polynomial in both the number of parameter values and the number of subroutines and functions.

Unfortunately, *it turns out to be doubly exponential in the level of the closure.* That is in the worst case. If the functions and predicates are commutative and associative and the program takes that into account, this can be improved to a mere exponential[2]. Practical experience with a real system trying to do this, even giving it knowledge about dimensions (*i.e.* it never adds meters to seconds, etc), shows that the run time at level 0 is okay, at level 1 is borderline, and at level 2 is ridiculous.

This shows that the program's power comes directly from its underlying representational primitives. If the concept under consideration lies at level 0, then the program will find it with no problem. If it lies at level 1, then the program must be heroic to find it, and at level 2 it simply will not find it.

My initial hope was that most relevant concepts would be near the surface, so to speak. That is, the underlying representational primitives would be so powerful that all concepts had expression as level 1 or less concepts. I subsequently found out that they could be made so only by giving the system highly specific and *ad hoc* seeming functions and subroutines.

One example of the system needing powerful primitives is in one relation needed in defining the gear-mesh rule in the Gear World Design Grammar. It turns out that the structure only works if the distance between the gear shafts is equal to the sum of the gears' radii. Now if the system has only the predicate EQUAL and the functions PLUS and TIMES, then this concept,

$$(r_1 \cdot r_2) \cdot (r_1 \cdot r_2) \quad (x_1 \quad x_2) \cdot (x_1 \quad x_2) \cdot (y_1 \quad y_2) \cdot (y_1 \quad y_2)$$

lies at a whopping level 3.

On the other hand, if the system has a DISTANCE function, then it is at only level one: $(r_1 \cdot r_2)$ DISTANCE (x_1, y_1, x_2, y_2).

Supposing the system has a DISTANCE function isn't so incredible; however, I found the system needing the predicate IS-TWICE-AS-BIG-AS (x, y) to make the concept of gear differential be level 1. If it has that one, why not IS-THRICE-AS-BIG-AS?

[2]For example, plus and times satisfy the

The final blow to this method is the realization that, often, a little semantic knowledge goes a long way. Specifically, suppose the system has only the function PLUS. The concept $y = 8x$ is at level 3:

$$y = ((x + x) + (x + x)) + ((x + x) + (x + x))$$

However, if the system knows one little technique, it can quickly find (or rule out) any linear function. It simply assumes the form $y = ax$ and uses the examples to get a number of equations. If $a = y/x$ is the same in all examples then it is found, otherwise it is ruled out. This is very much quicker than trying all combinations of additions of x and y to level 3.

Intuitively, it seems that there are many systems of representational primitives which exhibit regularities which may be exploited. Simple algebra is one example. Also, with some more domain knowledge, it is likely that almost all possibilities can be ruled out *a priori*. Using dimensions is an example where this is true. The difference between using dimensions and not using dimensions to constrain the number of combinations tried introduced a performance speedup of about a factor of 3 (from about 30000 to about 10000) on the small set of examples I was using.

There was one clever trick that came out of dealing with this algorithm. The trick used in the program actually made a major difference in the performance. This algorithm tends to produce lots of tautological relation-instances and useless constants, like (EQUAL x, x) and $(x + x - x - x)$. This can increase the number of relations to check and carry around by an order of magnitude. It is desirable to get rid of these.

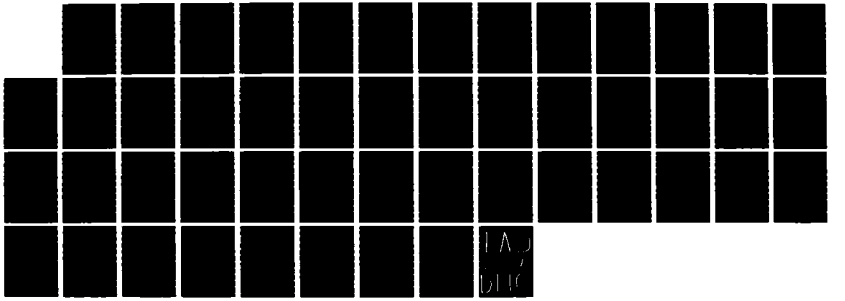
The clever way I found of doing this[3] is to try each relation and object expression on several *random* values. If the result always matches the values computed on the examples, then it is probably tautological or constant, so throw it away. It is necessary to make sure the random values always match the example's values, because the examples' value might be extremely rare. It is extremely unlikely, for example, to find a triple of integers x, y, z such that $x - y = z$. Therefore, it is likely that several random instantiations will all yield the same result. But if the examples satisfy the constraint, then the random values will not match the examples' values.

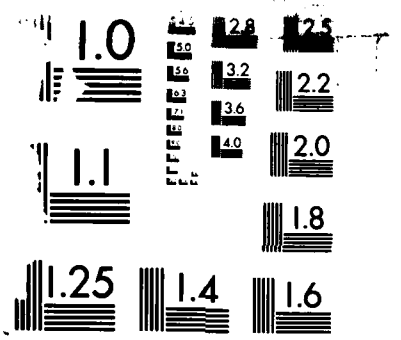
My conclusion is that the leveled closure method is to be used only as the very last resort, when no domain-specific knowledge is available. And then, the concepts had better be close to the surface of the representation language. An example of where this could conceivably be of use is in teaching a student: teach the student the relevant representational elements (like how to find the distance between points) and don't introduce the constraint-concept until all the representational elements are firmly in place. The only reason it works here is that in the right representation almost anything will work.

[3] I subsequently found out it is a technique sometimes used by mathematicians to decide if a theorem is worth working on.

AD-A174 730 LEARNING BY FAILING TO EXPLAIN(U) MASSACHUSETTS INST OF 2/2
TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB R J HALL
29 MAY 86 AI-TR-906 N00014-85-K-0124

UNCLASSIFIED F/G 5/10 NL





MICROCOPY RESOLUTION TEST CHART

Appendix D

Example Design Grammar

This Appendix contains the Design Grammars referred to in the text examples. The first section has the CMOS World Design Grammar, and the second has the Gear World Design Grammar.

D.1 CMOS World Grammar

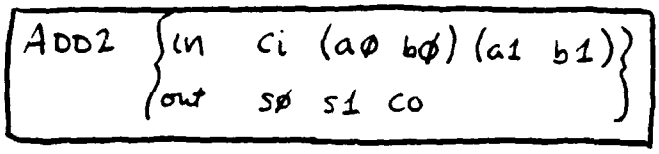
Rules with the same LHS are represented together and the LHS is represented only once. Rather than an entire graph, the LHS will be represented in the notation illustrated in Figure D.1.

For example, "in ci (a0 b0) (a1 b1)" indicates that there are five input variables with the names ci, a0, b0, a1, b1. Two variables in parentheses are neighbors via the same arc type; thus, they are interchangeable (commutative) inputs (or outputs).

In writing a RHS, if all inputs to a block are interchangeable, or if there is only one input, the arc types are left out. Similar treatment is afforded outputs. Data flows generally upward and to the right in these diagrams; if there is ambiguity, an arrowhead indicates the direction of data flow. Connection points are indicated by a black dot. Lines crossing without a dot at the intersection do not connect at that point.

Terminal graphs are shown in transistor notation.

(a)



(b)

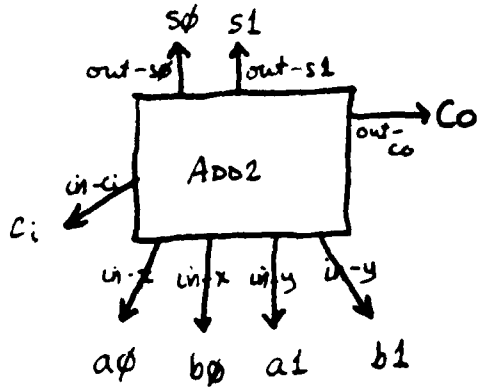
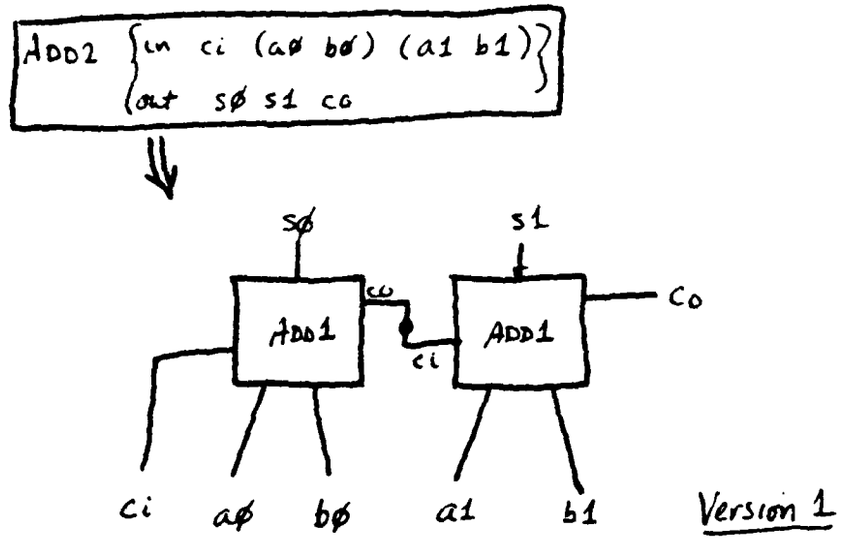
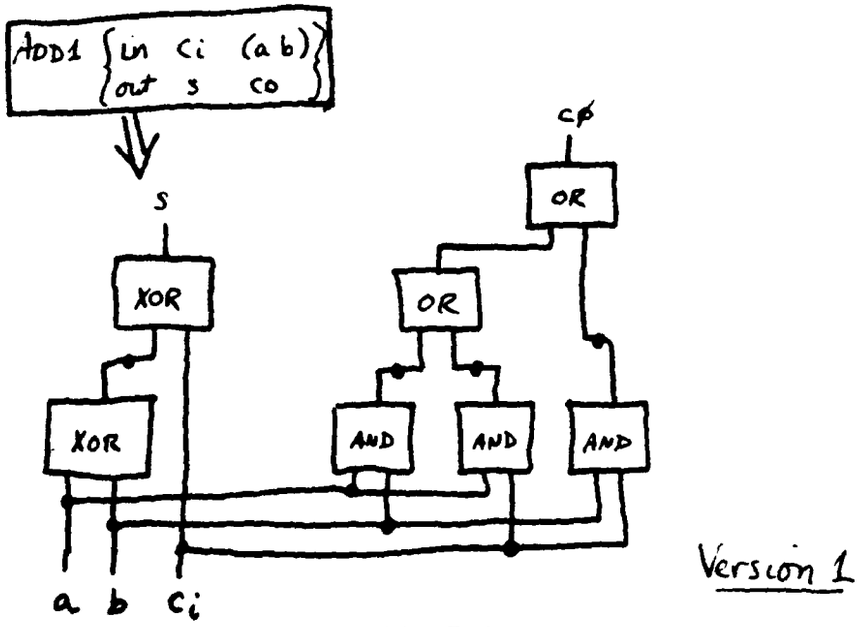


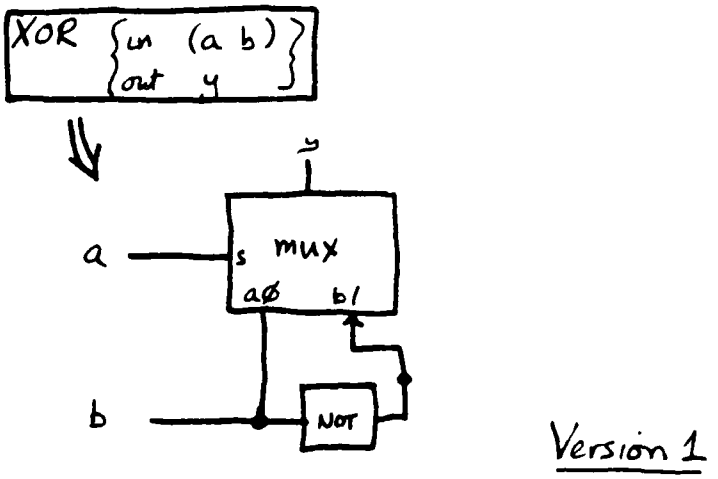
Figure D.1: (a) Special LHS Notation, (b) Meaning



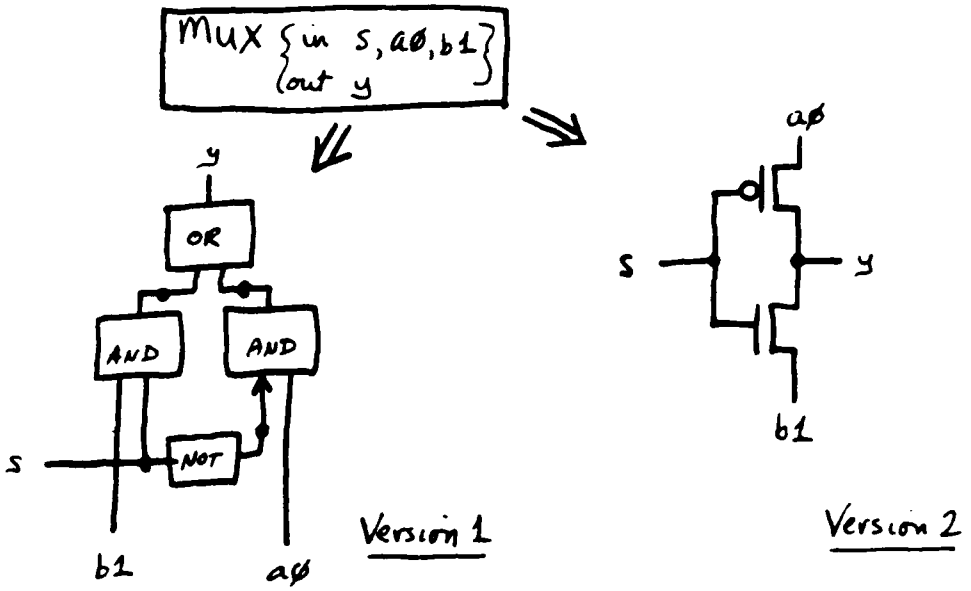
ADD2 Rule



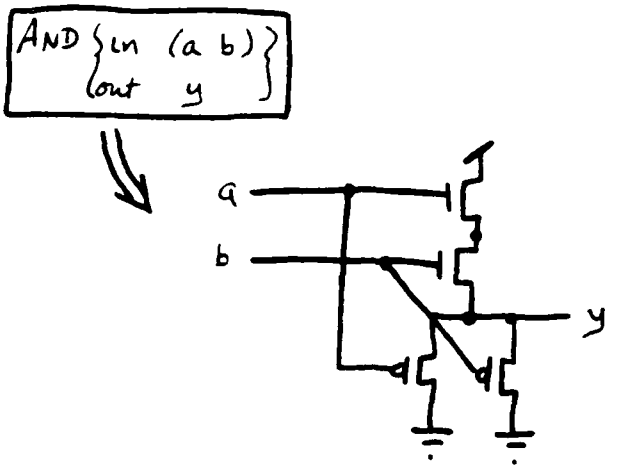
ADD1 Rule



XOR Rule

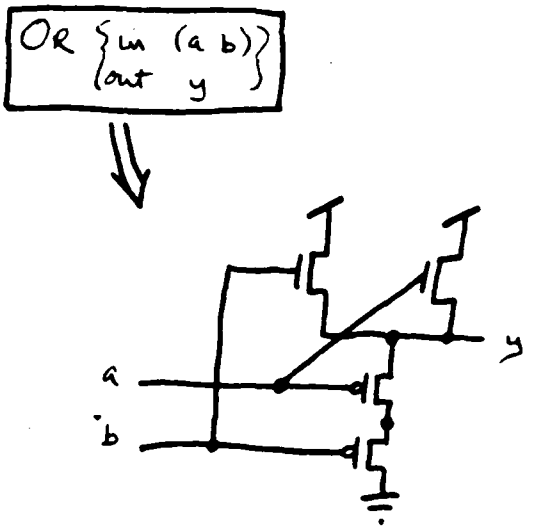


MUX Rules



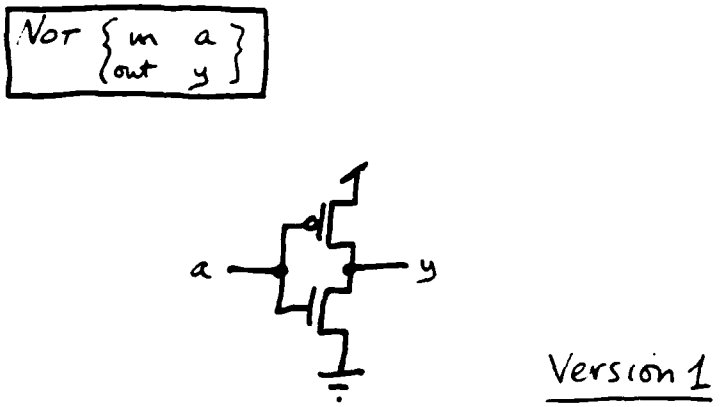
Version 1

AND Rule

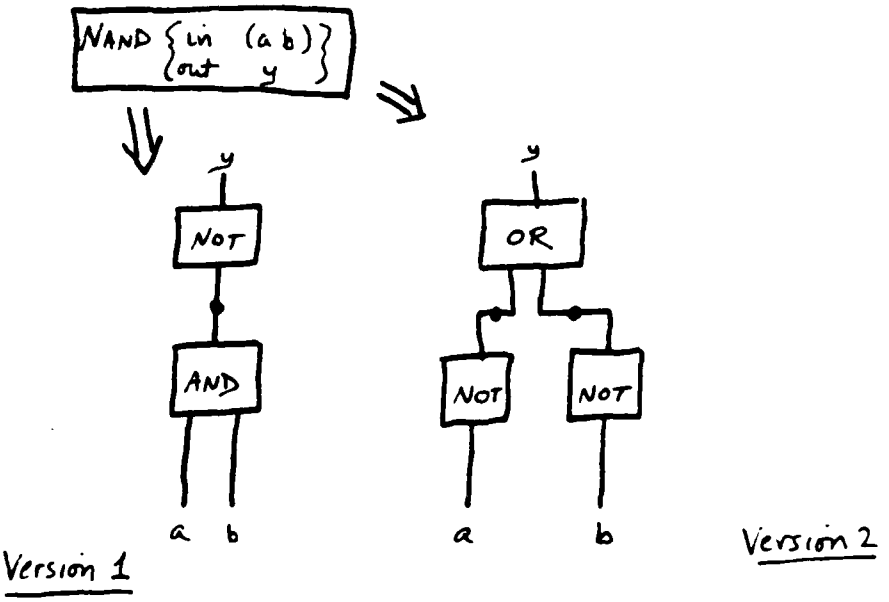


Version 1

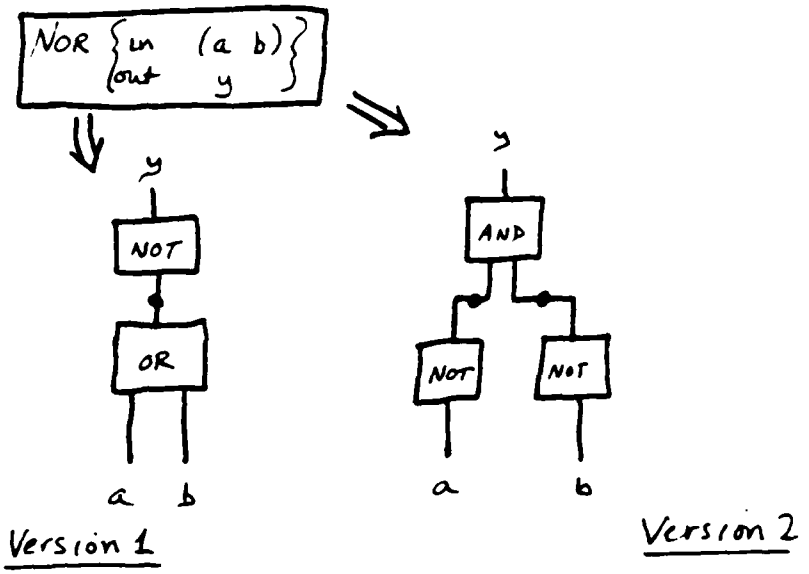
OR Rule



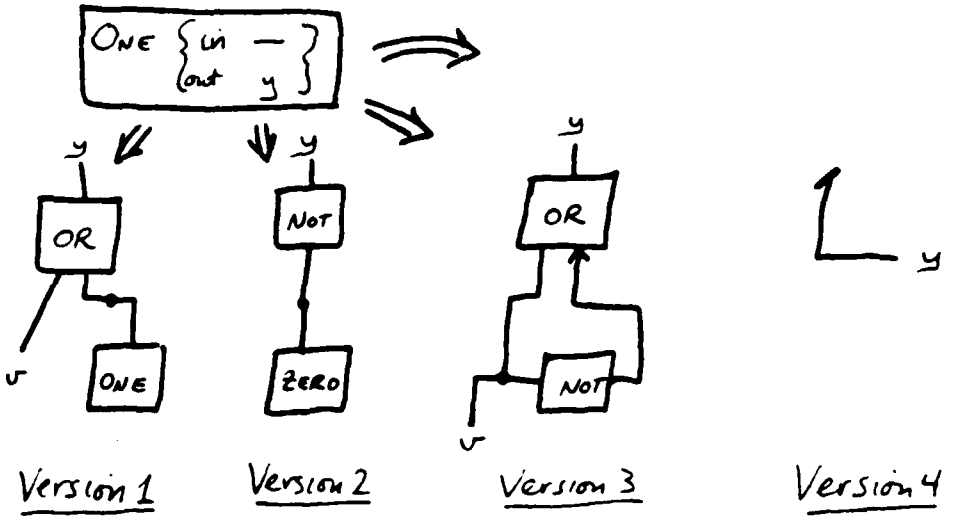
NOT Rule



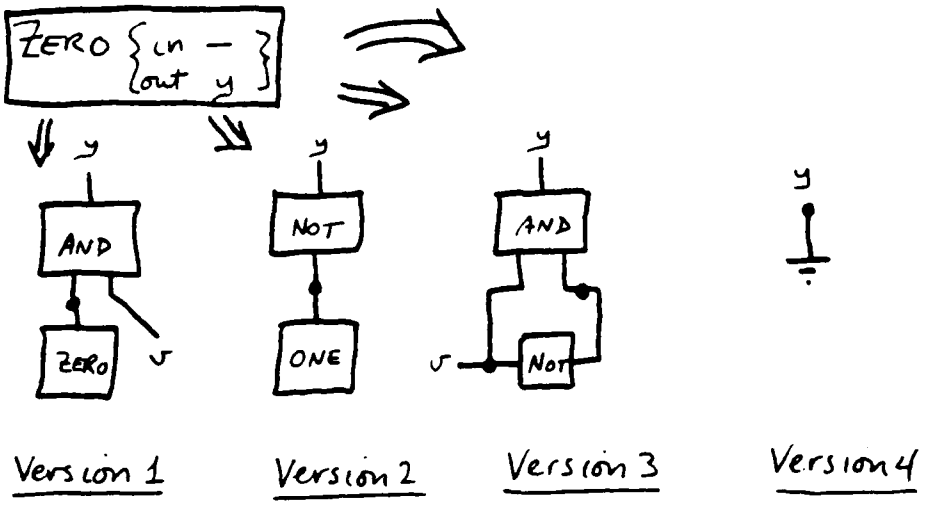
NAND Rules



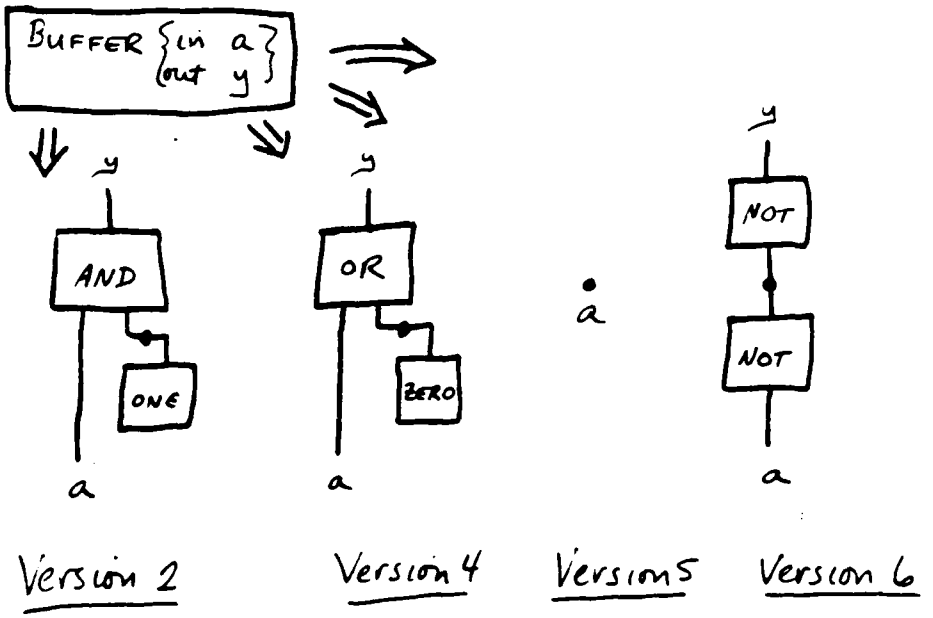
NOR Rules



ONE Rules



ZERO Rules

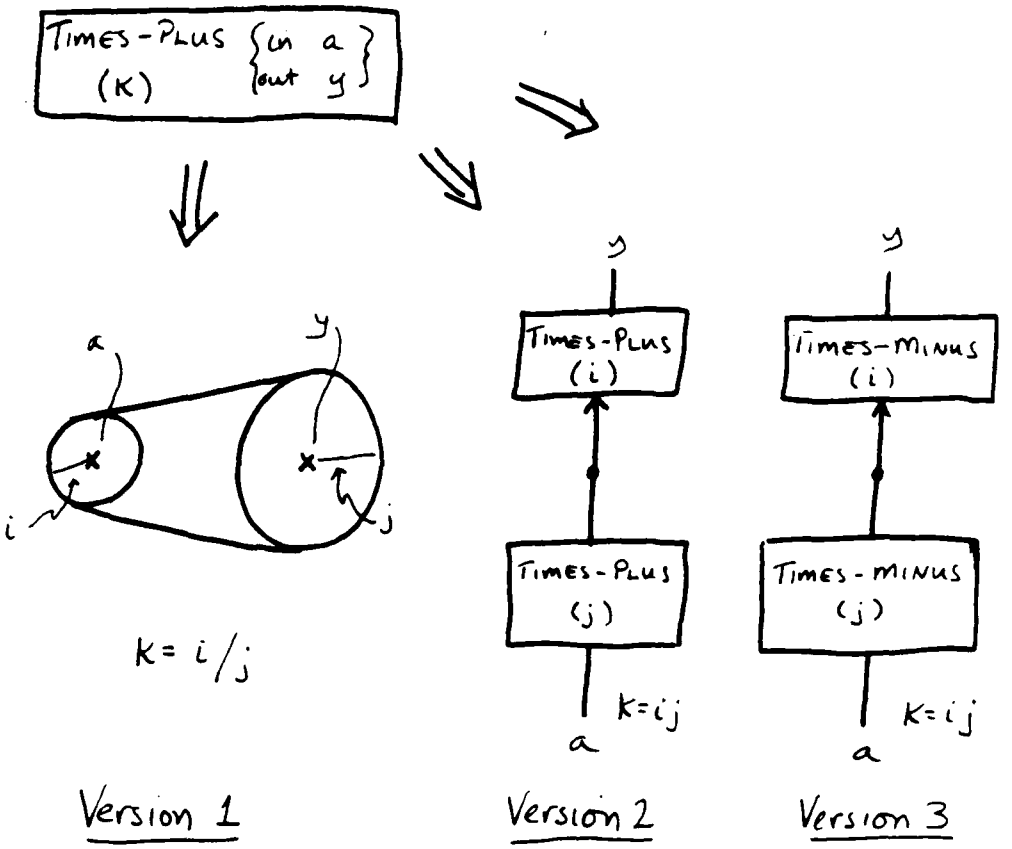


BUFFER Rules

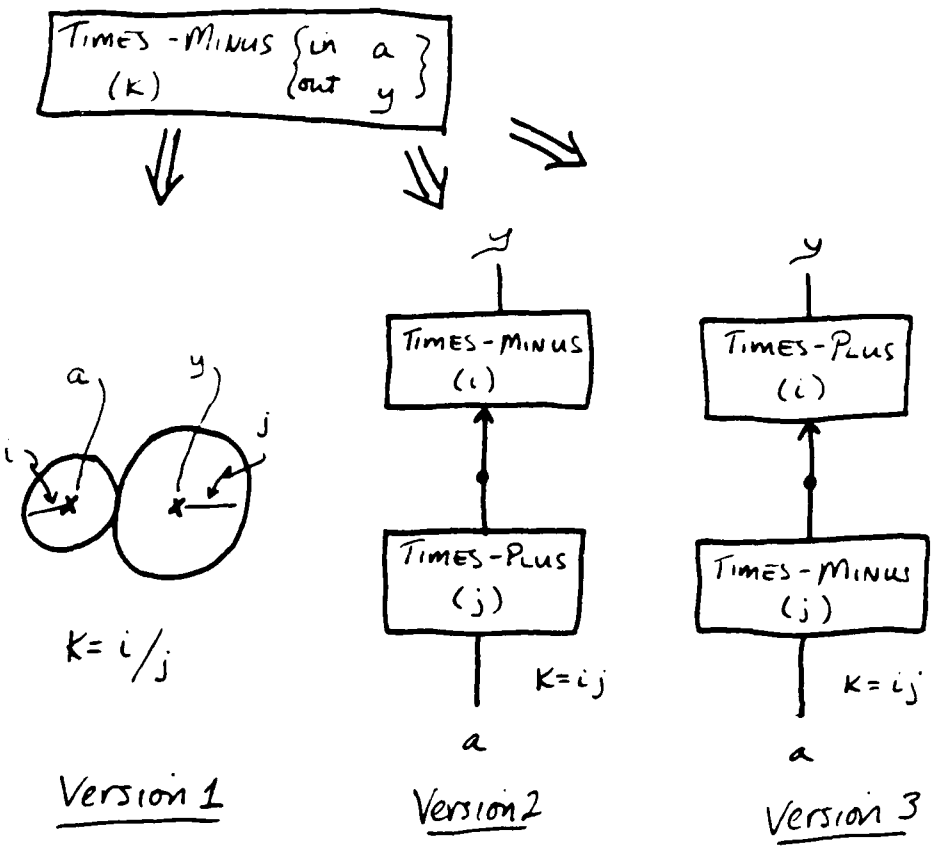
D.2 Gear World Grammar

As the elements in GW have parameters, parameters are shown as lower case letters and relations are indicated by writing them below the RHS. The non-terminals have parameters also: these are put in parentheses next to the non-terminal name.

Terminal graphs here are drawn schematically instead of in graph notation. Circles indicate either sprockets or gears. A line wrapping two circles indicates a chain wrapping two sprockets. A small x indicates a shaft to which the gear or sprocket is mounted.



TIMES-PLUS Rules



TIMES-MINUS Rule

Appendix E

The System and Some Actual Examples

The purpose of this Appendix is to give a sense of the strengths and weaknesses of the current implementation of the Analysis/Precedent Analysis algorithm. The original implementation is documented in [5]. It had some major weaknesses which served to obscure some issues. However, the second generation, on which these examples were run, is much improved.

E.1 The Implementation

The original implementation had demonstration routines for all of the design competences, as well as Precedent Analysis and Rule Re-analysis. The re-implementation focused on the analysis algorithm itself, as this is the heart of the entire system. Rule Re-analysis has not been re-implemented as yet, mostly because the system's database design has not been completely specified. From here on, "the system" shall refer to the re-implementation of the Analysis/Precedent Analysis algorithm and its supporting code.

The entire system, except the graphics, is implemented in Common Lisp [18] on a Symbolics 3600. Its architecture is founded on a package of utilities which implement a *graph* data type. One may represent any labeled digraph, except those with more than one arc of a given type between the same pair of nodes. Among the primitive operators for graphs are creation, combination, surgery, node and arc deletion, subgraph isomorphism, and quotient by a vertex equivalence relation. There are no side effects in any of the operations; any graph resulting from an operation on one or more graphs is completely disjoint from the input graphs. There is an auxiliary data structure, called a *gmap*, which maps nodes from one graph to single nodes of another. Gmaps represent such things as subgraph isomorphisms, and the partial matches built by the Analysis algorithm. The subgraph isomorphism algorithm is described in Appendix B.

E.2 Experiments

To describe an example, I will describe the inputs given to the system, the output (partial) derivation, the rule which is generated (if the example is not completely explained), and some statistics. The inputs are the Design Grammar with which the system starts the run, the pair of graphs which make up the precedent, and the *SEARCH-DEPTH* parameter (the maximum lookahead depth referred to in Chapter 3). In the diagrams, the left hand graph is considered to be the high level description. The diagrams indicate the current state of the partial match by highlighting the nodes which correspond. Again, the diagram is slightly ambiguous in that it does not precisely indicate the exact correspondence. It should be obvious however.

The statistics include the approximate real time duration of the run and the *progress history:* a list of the numbers of derivation steps between successive discoveries of progress. For example, (2,6.2) indicates that in the ten step partial derivation the system found progress after the second, eighth, and tenth steps. If the derivation is partial, the last term will appear parenthesized to indicate that, instead of progress having been found, the graph resulting is the smallest graph considered in the last, unsuccessful round of search. (See Section 4.2.1 for a discussion of this criterion.)

E.2.1 MUX-0 (Depth First)

The original implementation of this algorithm used depth-first lookahead instead of the breadth-first strategy described here. This has a severe drawback: it is possible to waste much time finding unnecessarily long and inefficient derivations. The reason is that the system might try a completely arbitrary rule application first, then apply the correct rule, making progress in two steps when it could have made the same progress using only one derivation step. Since the algorithm is greedy, it takes the long derivation as soon as it finds it. The following example illustrates.

(This was produced with an older version of the system, so only the initial and final graphs have both graphs shown. The intervening derivation steps show only the current state of the left-hand graph.)

- **Initial grammar:** Appendix D
- **Search Depth:** 4
- **Nodes Searched:** 59
- **Time:** about 2 minutes
- **Progress History:** (2,4)

Note that the first and third steps are inverses, and hence could be left out.

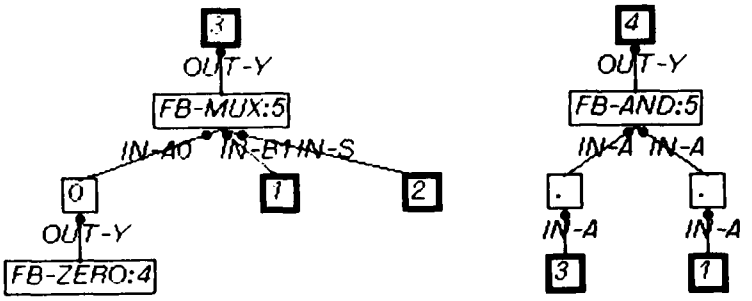


Figure E.1: The Input Precedent

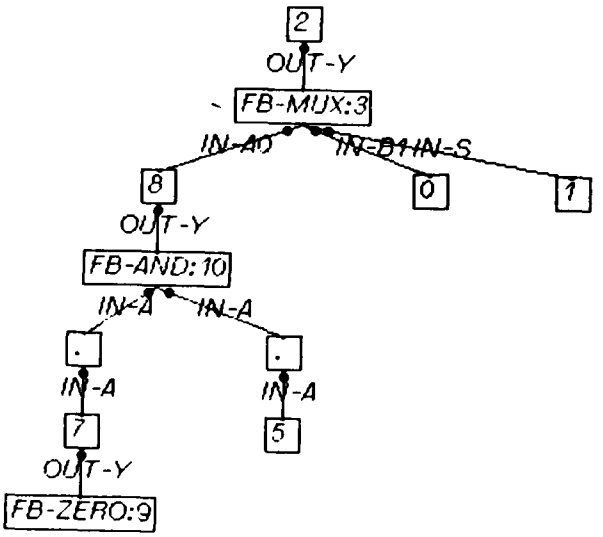


Figure E.2: After One Step

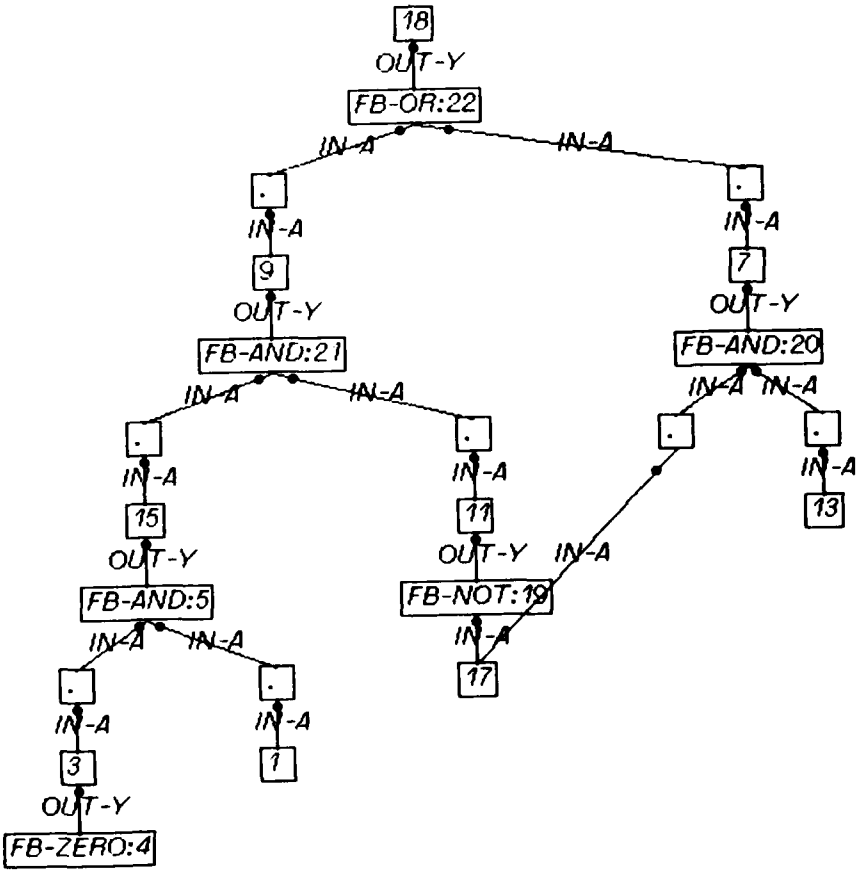


Figure E.3: After Two Steps

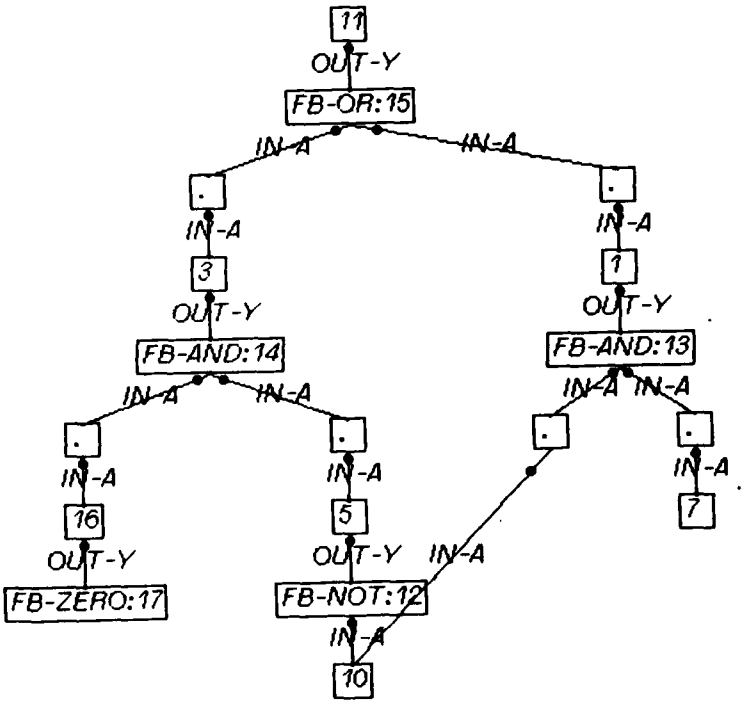


Figure E.4: After Three Steps

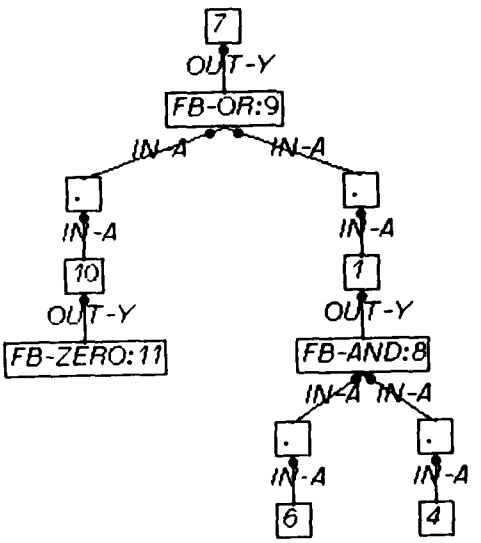


Figure E.5: After Four Steps

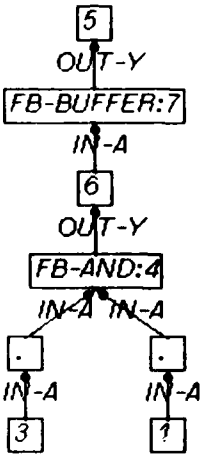


Figure E.6: After Five Steps

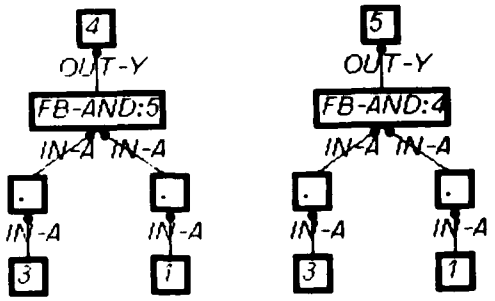


Figure E.7: After Six Steps

E.2.2 MUX-0 (Breadth First)

The same example was run on the breadth-first system. (All further examples are run breadth-first.) The derivation produced was the same as that of the depth-first system, except that the first and third steps are left out.

- **Initial grammar:** Appendix D
- **Search Depth:** 3
- **Nodes Searched:** 27
- **Time:** about 30 seconds
- **Progress History:** (1, 3)

E.2.3 Effect of *SEARCH-DEPTH*, I

This Section and the next demonstrate the effect of lookahead search-depth on power of the algorithm. The run in this Section was done with *SEARCH-DEPTH* at 2. The result was that the system failed to find a complete explanation and deduced the rule: NOT(NOR(x, y)) $\equiv$ OR(x, y).

- **Initial grammar:** Appendix D
- **Search Depth:** 2
- **Nodes Searched:** 21
- **Time:** about 1 minute
- **Progress History:** (1, 1, 2, (2))

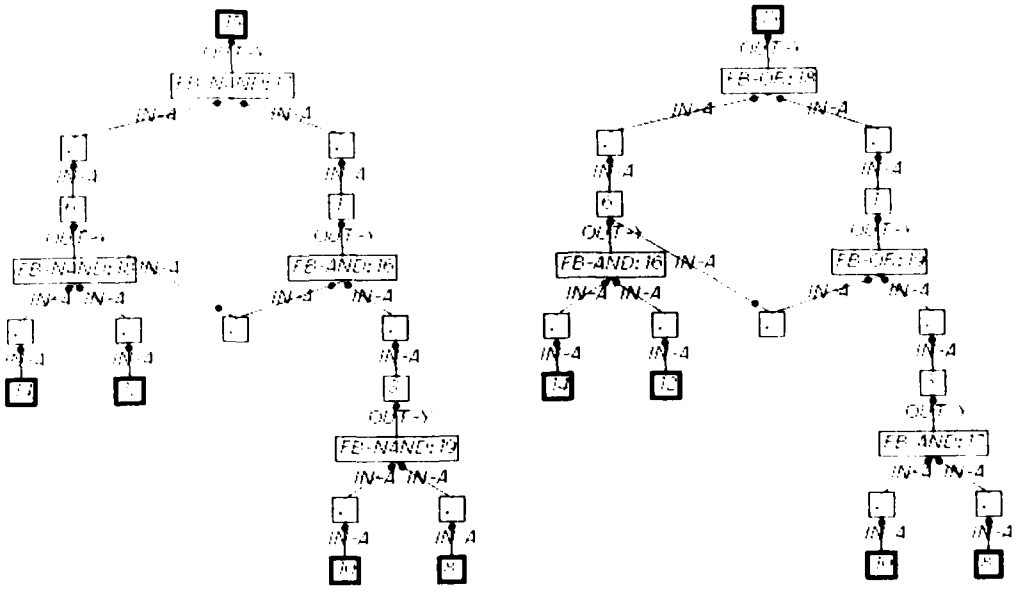


Figure E.8: Precedent

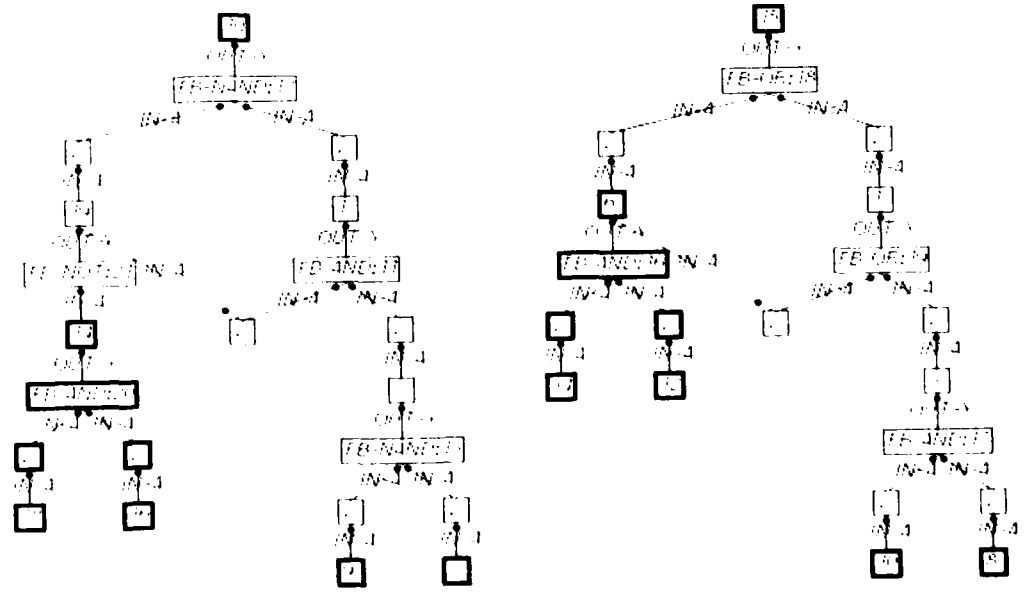


Figure E.9: After One Step

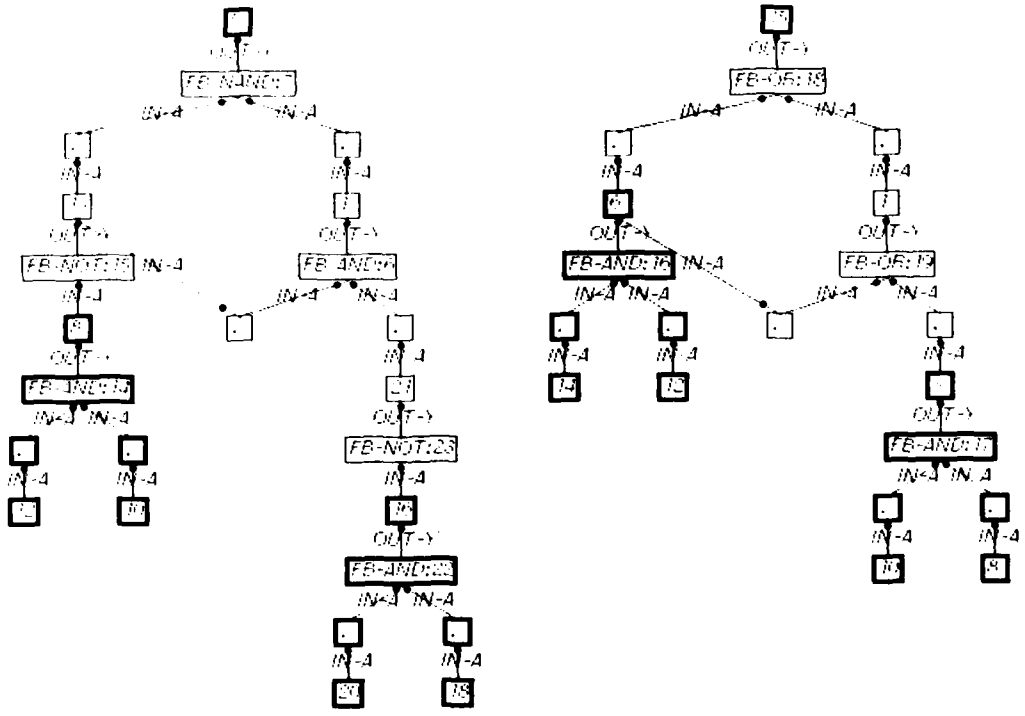


Figure E.10: After Two Steps

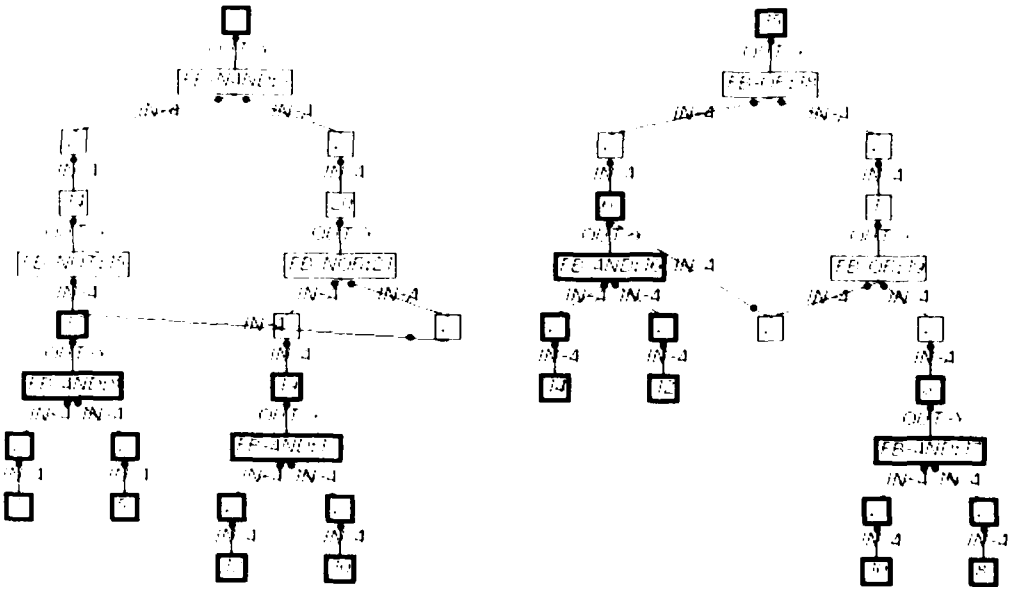


Figure E.11: After Three Steps

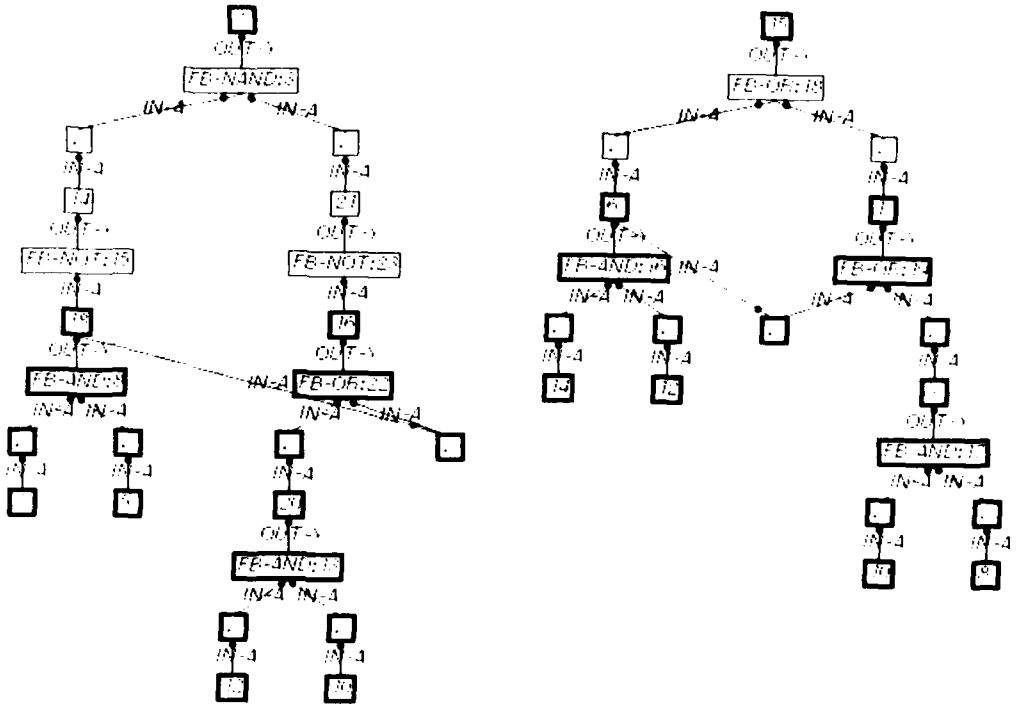


Figure E.12: After Four Steps

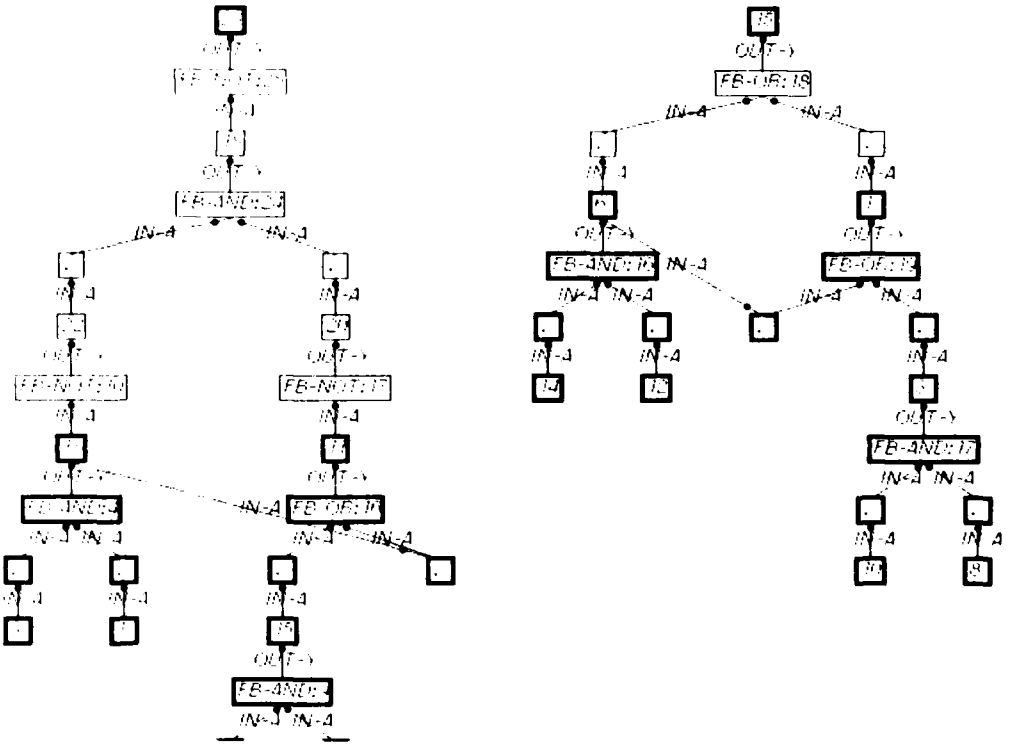


Figure E.13: After Five Steps

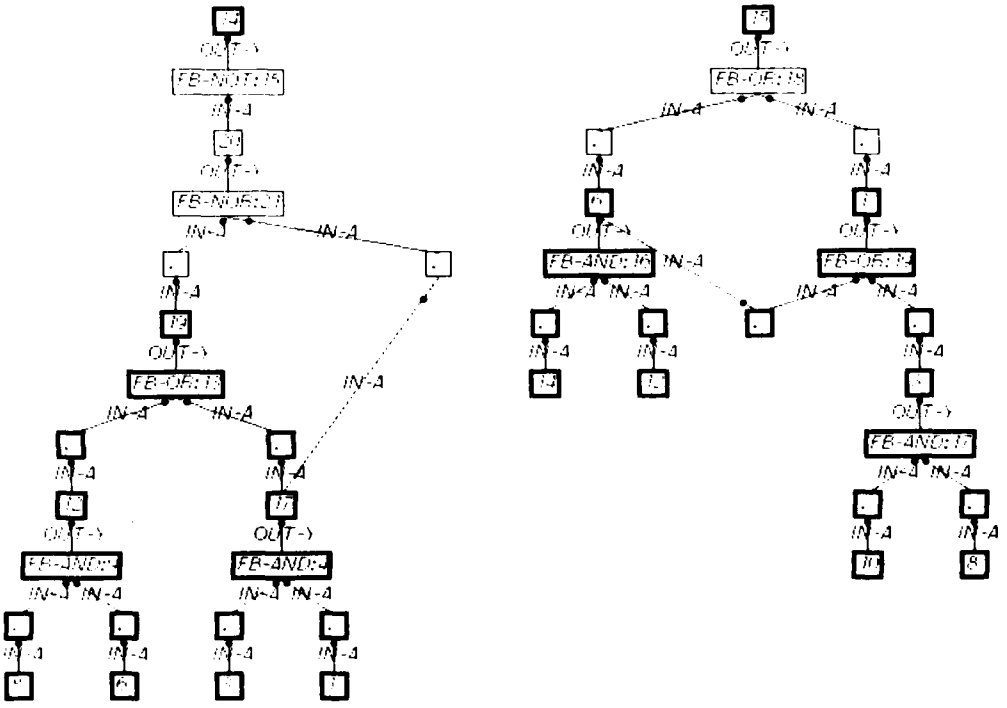


Figure E.14: After Six Steps

E.2.4 Effect of *SEARCH-DEPTH*, II

The same example as in the previous section was done with *SEARCH-DEPTH* increased from 2 to 3. This resulted in the system completely explaining the precedent.

- **Initial grammar:** Appendix D
- **Search Depth:** 3
- **Nodes Searched:** 31
- **Time:** about 2 minutes
- **Progress History:** (1.1.2.3.2)

The first four steps of this result derivation were the same as the first four of the previous derivation, so the following sequence of steps starts after the fifth step.

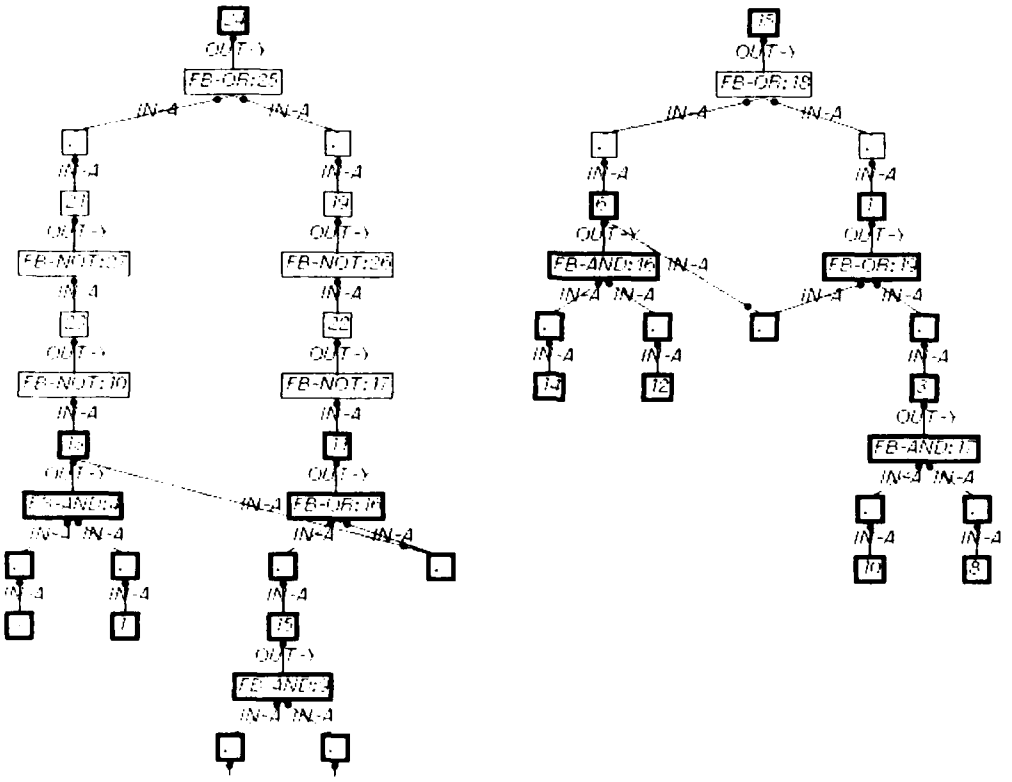


Figure E.15: After Five Steps

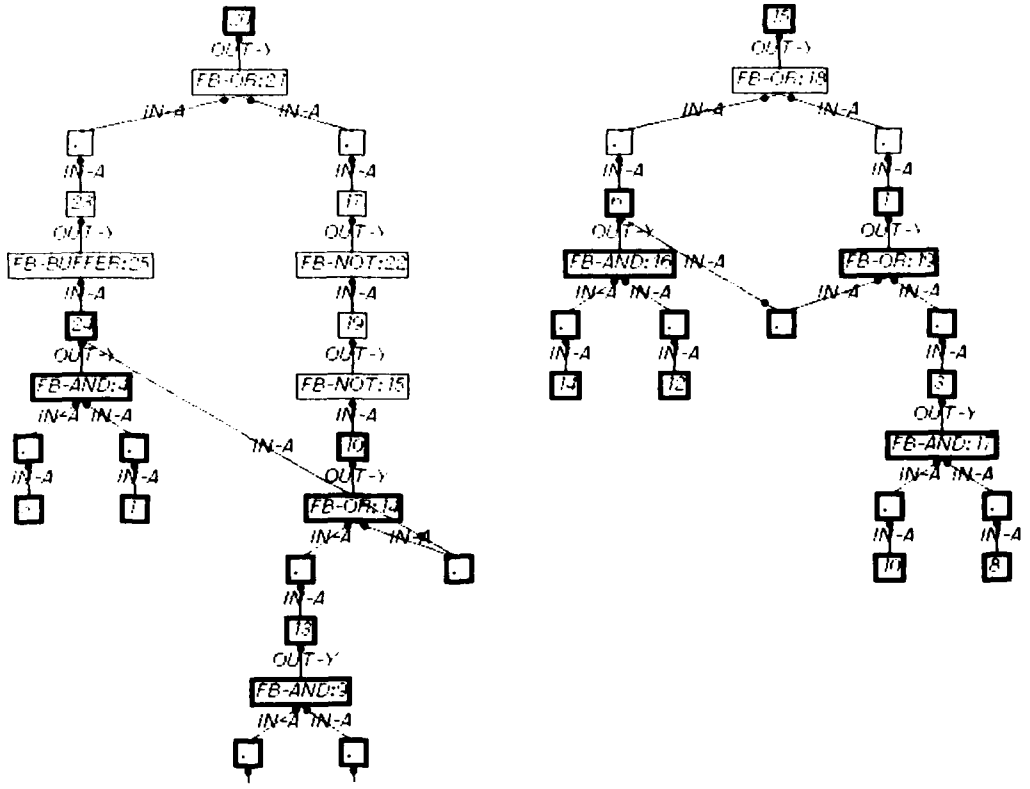


Figure E.16: After Six Steps

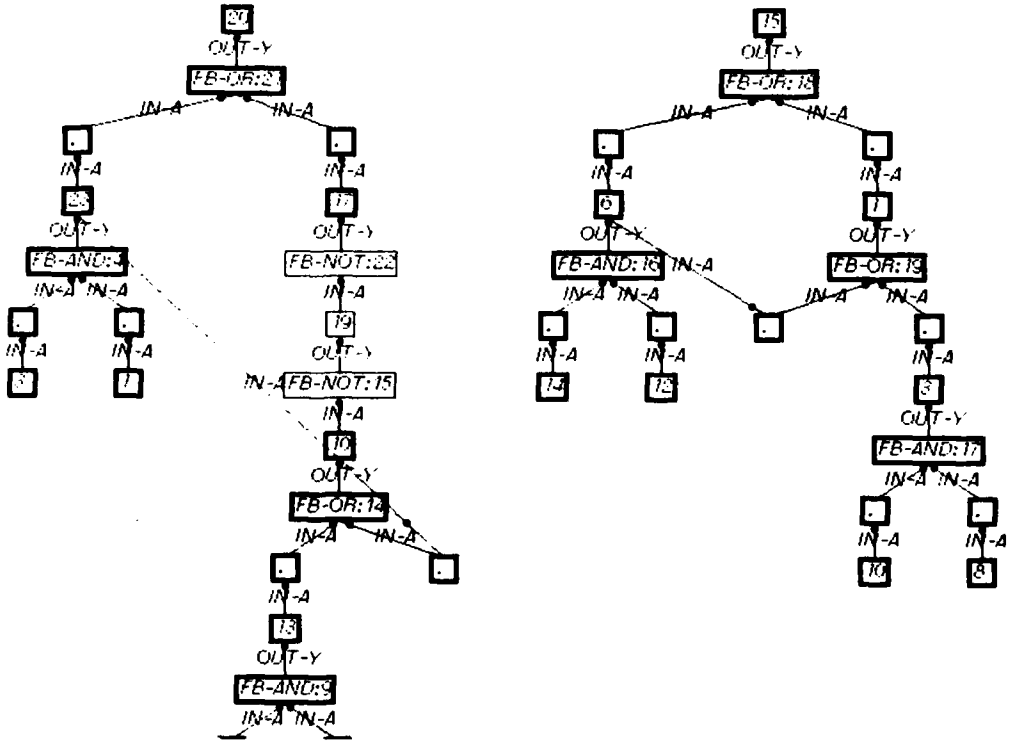


Figure E.17: After Seven Steps

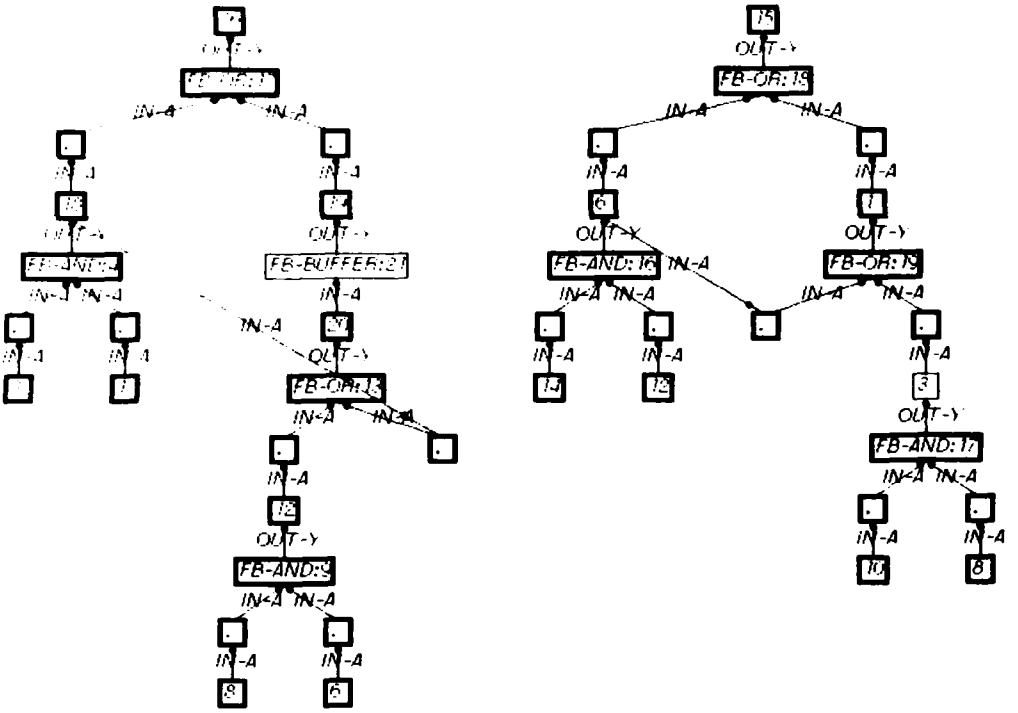


Figure E.18: After Eight Steps

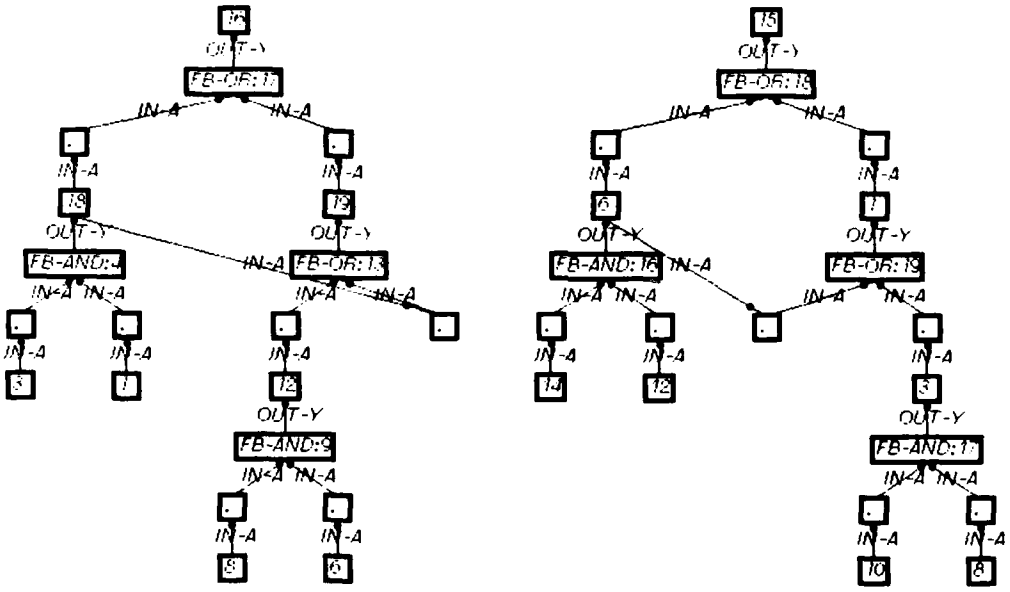


Figure E.19: After Nine Steps

E.2.5 Effect of the Grammar, I

The example of this Section (XOR, with one input tied to 0) was attempted with the Grammar of Appendix D, but the run time was prohibitively long ($\gg$ 2 hours, hundreds of nodes searched). There are a few reasons for this: the minimum search depth required to find the derivation is 8; because of the ZERO $\rightarrow$ NOT (ONE) $\rightarrow$ NOT (NOT (ZERO)) cycle the search tree has a large branching factor; the intermediate graphs become large, so that matching the RHS of the ADD1 rule to the large graphs requires a long time.

Thus, it became of interest to vary the input grammar. The variation chosen was to prohibit the system from trying certain of the rule directions at all. Define *GRAMMAR1* as the grammar of Appendix D, but where the ZERO and ONE rules are never allowed in the forward direction, and the ADD1 rule is never allowed in reverse.

- **Initial grammar:** GRAMMAR1
- **Search Depth:** 8
- **Nodes Searched:** 42
- **Time:** about 1 minute, 30 seconds
- **Progress History:** (8)

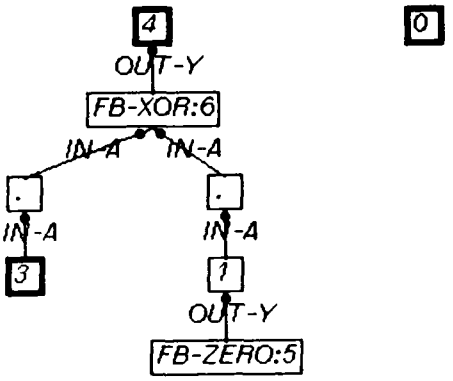


Figure E.20: Precedent

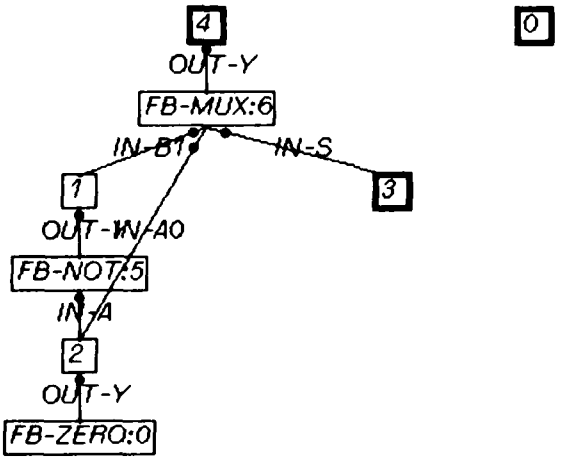


Figure E.21: After One Step

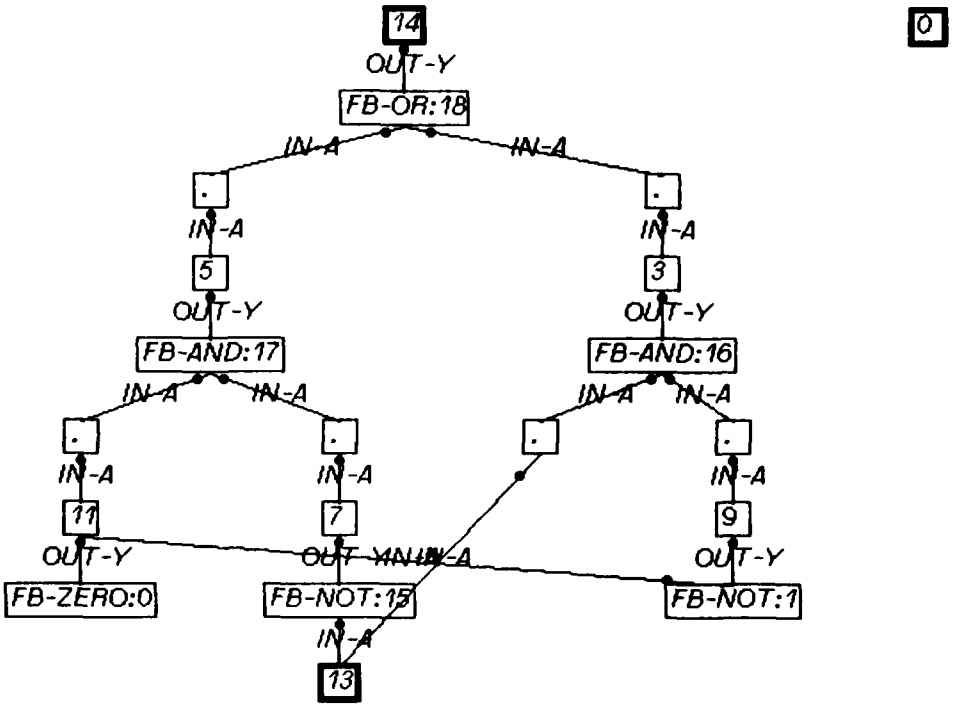


Figure E.22: After Two Steps

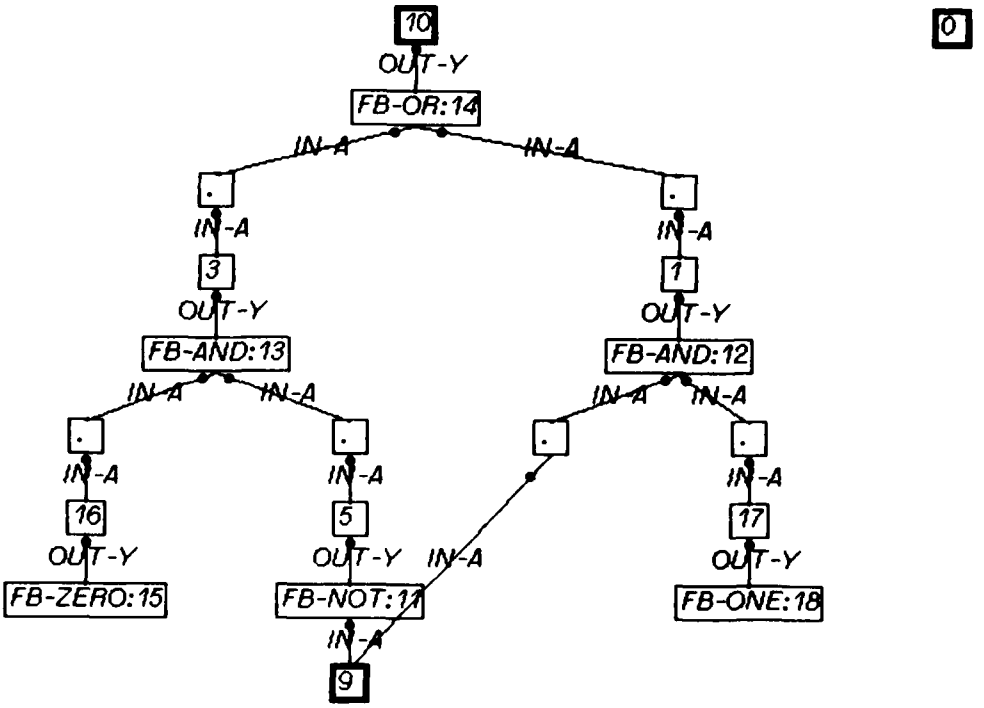


Figure E.23: After Three Steps

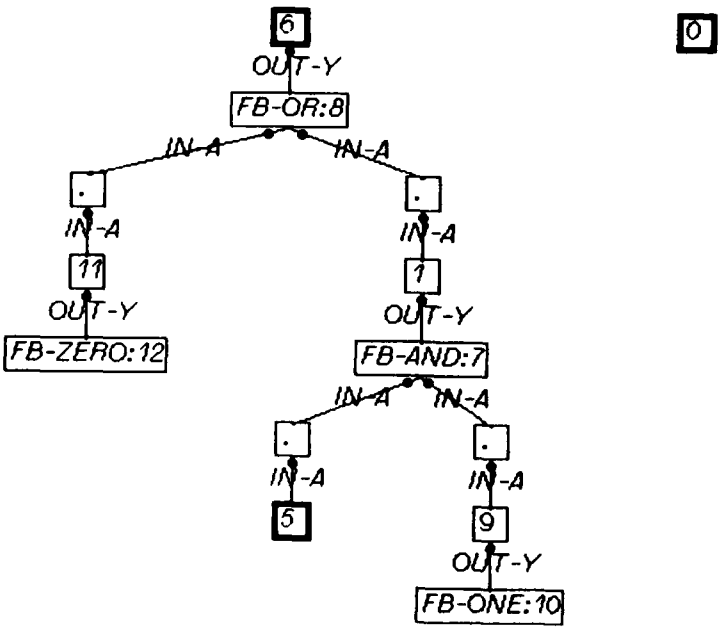


Figure E.24: After Four Steps

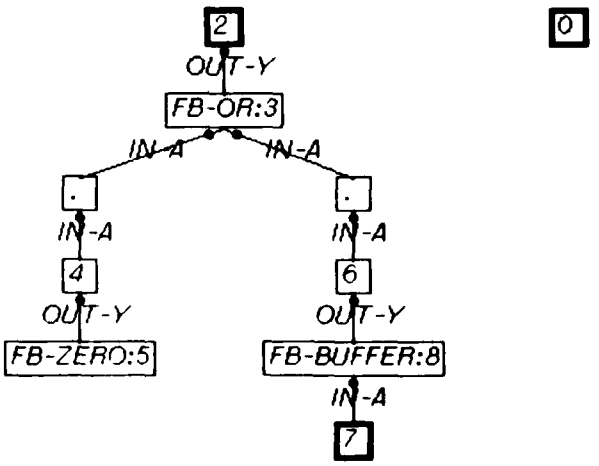


Figure E.25: After Five Steps

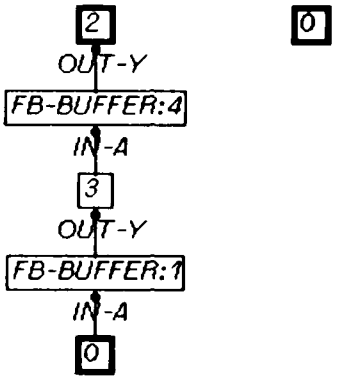


Figure E.26: After Six Steps

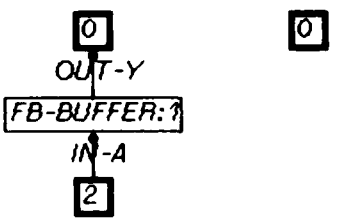


Figure E.27: After Seven Steps

Figure E.28: After Eight Steps

E.2.6 Effect of the Grammar, II

The previous section illustrated that having too many relatively useless rules available can cause a catastrophic increase in search. This section will illustrate that an inability to focus attention to a small portion of the design can have the same effect.

Consider the precedent in Figure E.29. This is the two-bit incrementer circuit with carry output. A straight-forward attack by the system using either Appendix D's grammar or GRAMMAR1 would in principle succeed. Unfortunately, the search takes far too long. It was not run to completion, but a crude estimate is that it would take at least 12 hours and explore over 10000 search nodes. (The resulting derivation has 41 GRAMMAR1 steps.)

The reason is not that there were too many rules around, but that there were too many ways to make progress! That is, to optimize the circuit, one could first simplify the low order bit, then do the high order bit. Also, in simplifying each bit slice, one can simplify the XOR part or the carry circuitry. *The analysis algorithm, with its breadth first approach, effectively does all at once.* One can readily see that all of the possible attacks do not combine linearly. The size of the search tree is roughly k^n for some constant k. If there were four disjoint methods of attack on the problem, there would be k^{4n} search nodes, as opposed to $4k^n$.

The system can deal with this problem by using derived rules. If the system explains a precedent, then it may use that equivalence as a rule. This makes both the effective path length of the derivation and the maximum search depth small. The system was asked to derive the following three rules based on GRAMMAR1:

- XOR $(0, a) \equiv a$
- XOR $(1, a) \equiv$ NOT (a)
- MUX $(s = 1, b1 = x, a0 = y) \equiv x$

Using those in addition to GRAMMAR1, the system was able to derive the one-bit incrementer rule: (out-co out-s) ADD1 (x, 1, ci= 0) = (x, NOT (x))

This last derivation required about 20 minutes, searching 123 search nodes. The progress history was $(2, 2, 8)$.

Define *GRAMMAR2* to be GRAMMAR1 together with these rules. For brevity, the last six steps of the derivation have been left out.

- **Initial grammar:** GRAMMAR2
- **Search Depth:** 6
- **Nodes Searched:** 19
- **Time:** about 1 minute
- **Progress History:** (2.1.1.6)

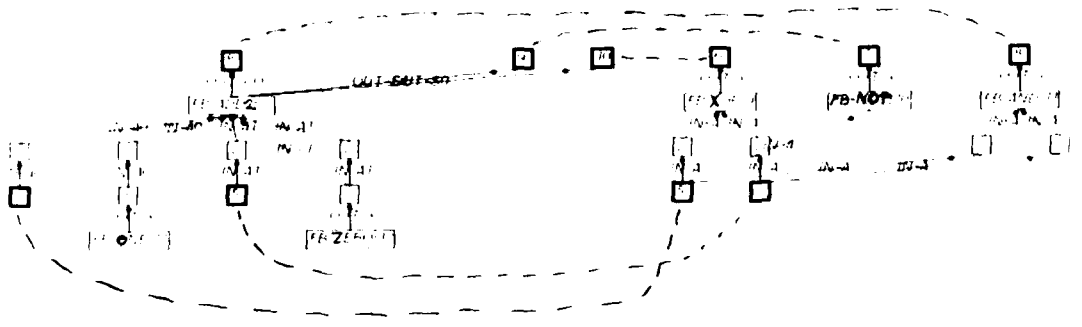


Figure E.29: Two-bit Incrementer Precedent

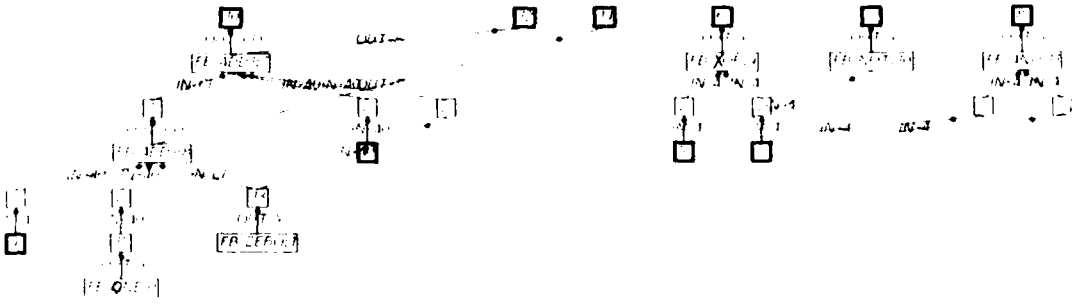


Figure E.30: After One Step

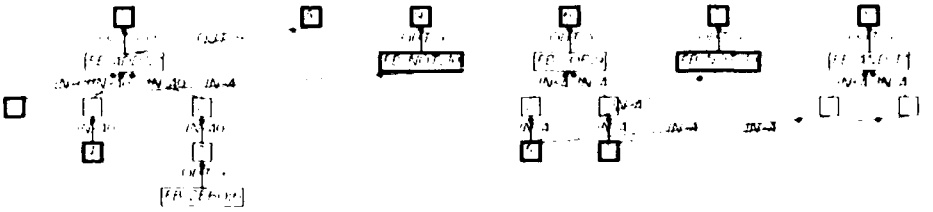


Figure E.31: After Two Steps

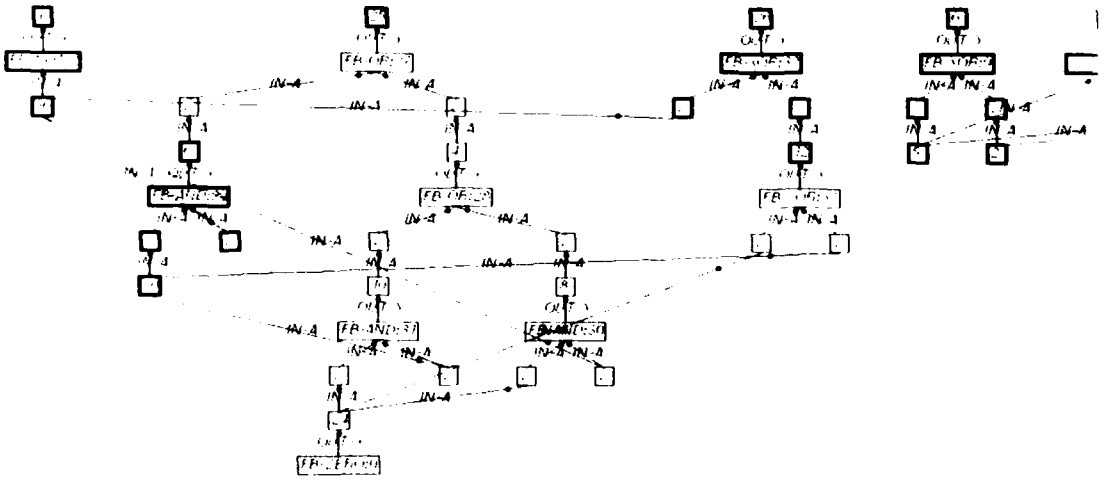


Figure E.32: After Three Steps

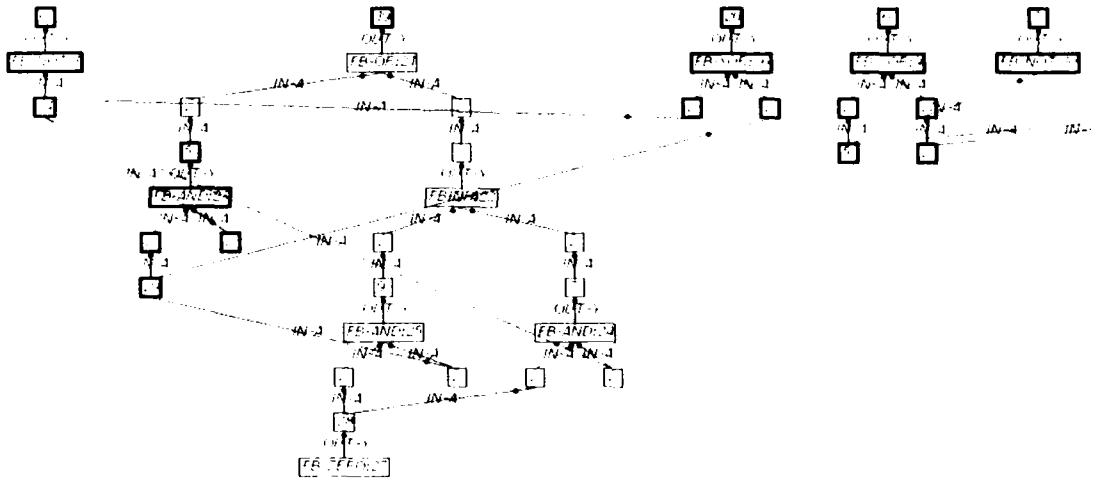


Figure E.33: After Four Steps

E.2.7 Greed

The greed of the algorithm can be a detriment at times. This problem is much worse for the depth-first version of the algorithm, but it can also hurt the breadth-first. An example of this is in the trivial case of deriving a derived rule from a grammar which already contains it. For example, suppose we add the XOR $(1, x) \equiv$NOT (x) to GRAMMAR1. Then pose the same equivalence as a precedent. If the system happens to pick the derived rule first, then it will find progress and stop, as one hopes. On the other hand, if it finds the expansion of XOR into (MUX and NOT) first, it will do that, since the NOT represents progress. The derived rule is no longer applicable, so the system will have to continue from there and rederive the entire rule!

This problem arose when the system was attempting to derive the one-bit incrementer precedent mentioned in the previous section. At some point, an XOR block had one input tied to ONE. The grammar had the appropriate derived rule for this case, but the system chose the XOR expansion first. It then would have had to rederive the XOR-ONE rule. This is why the system was also given the MUX-ONE rule to derive and use.

Note that the MUX-ONE derivation is a postfix of the XOR-ONE derivation. Thus, a STRIPS-like "triangle-table" setup might alleviate this problem.

E.2.8 The Scenario Example

With these insights, we are ready to attack the scenario example. See Figure E.34. As it stands, it is much too complex to be attacked using only the grammar of Appendix D. The full story is shown below. (Note that the search depth parameter was set higher than necessary. This probably inflated the real time figure by about 5 to 10 minutes, by causing the system to search seven levels on the last round instead of five.)

- **Initial grammar:** GRAMMAR1, plus the XOR-0, XOR-1, MUX-1 rules; AND $(a, a) \equiv a$, NOT (NOT a) $\equiv a$, AND $(a, 1) \equiv a$, OR $(a, 0) \equiv a$; all BUFFER rules removed.
- **Search Depth:** 7
- **Nodes Searched:** 808
- **Time:** about 95 minutes
- **Progress History:** $(2, 5, (2))$

Steps three through five are left out for brevity.

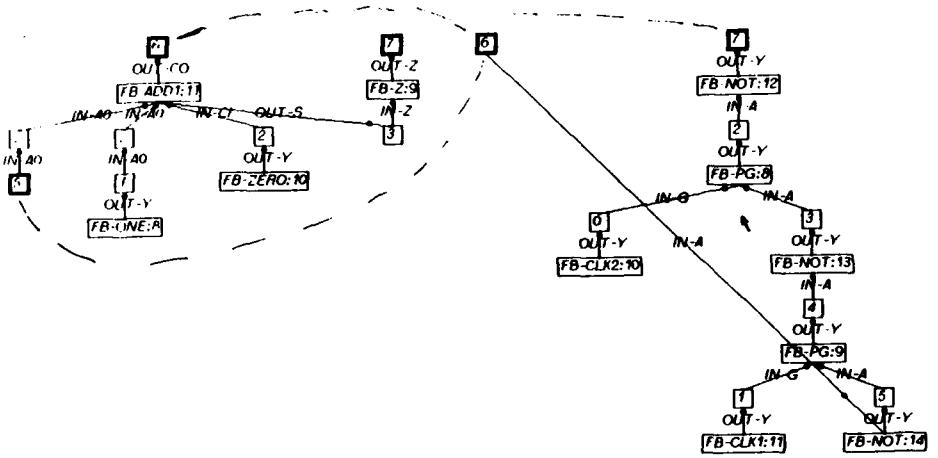


Figure E.34: Scenario Precedent

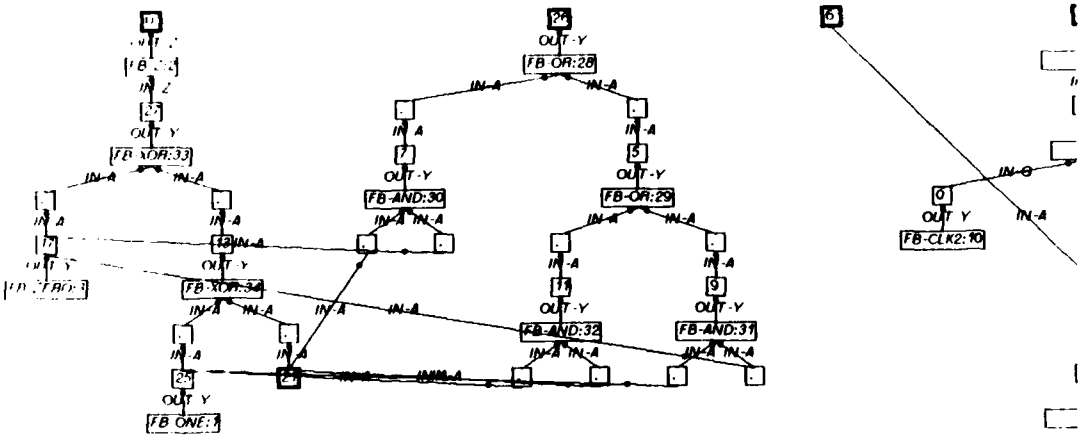


Figure E.35: After One Step

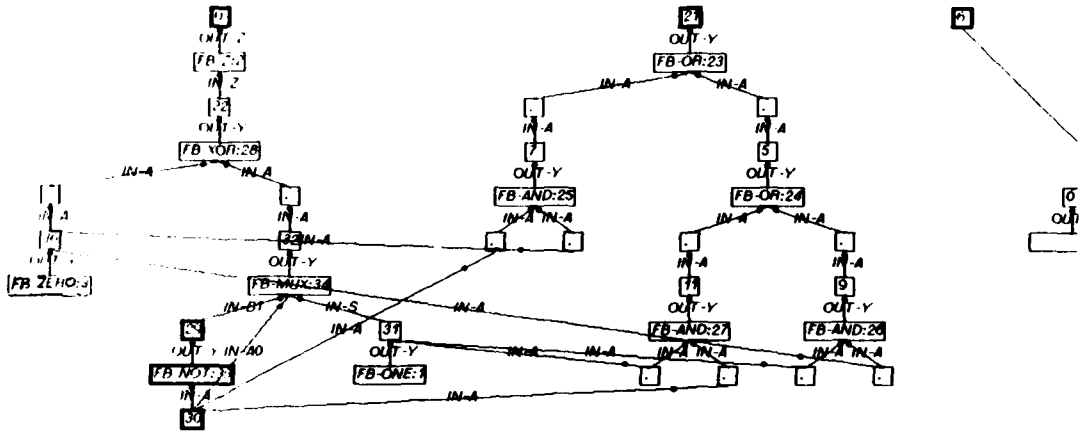


Figure E.36: After Two Steps

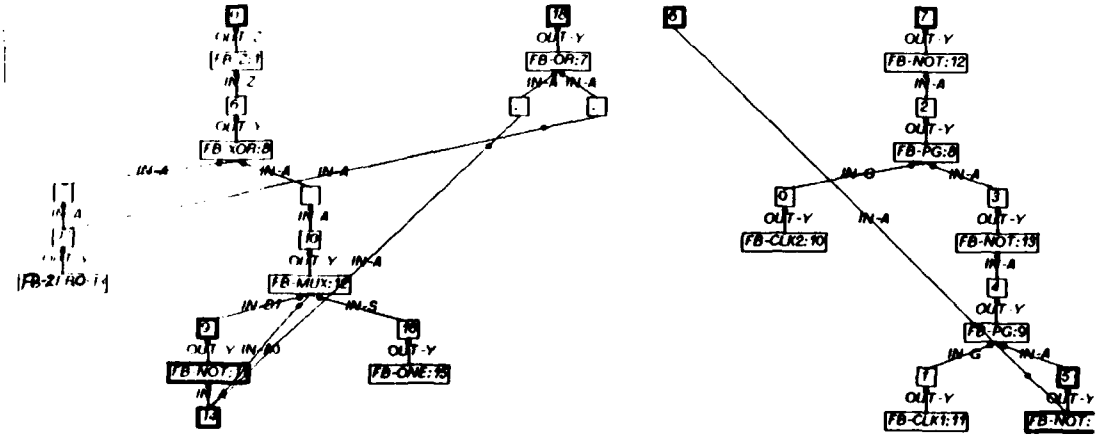


Figure E.37: After Six Steps

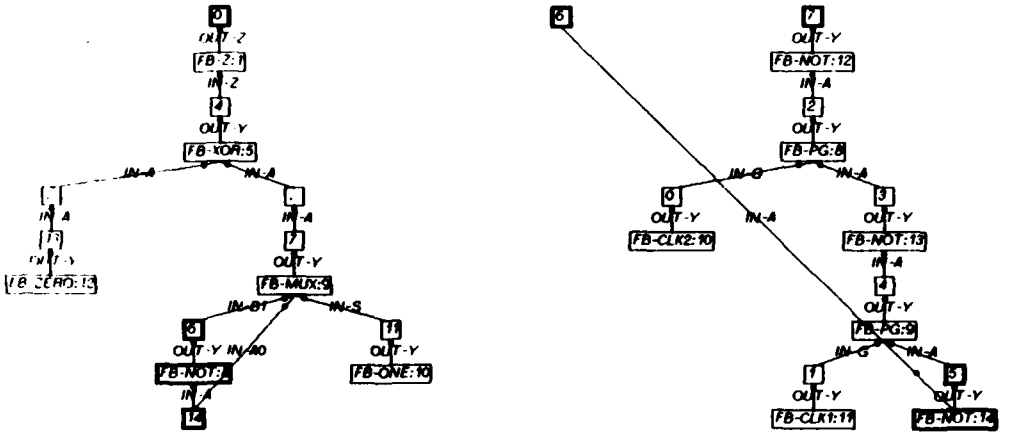


Figure E.38: After Seven Steps

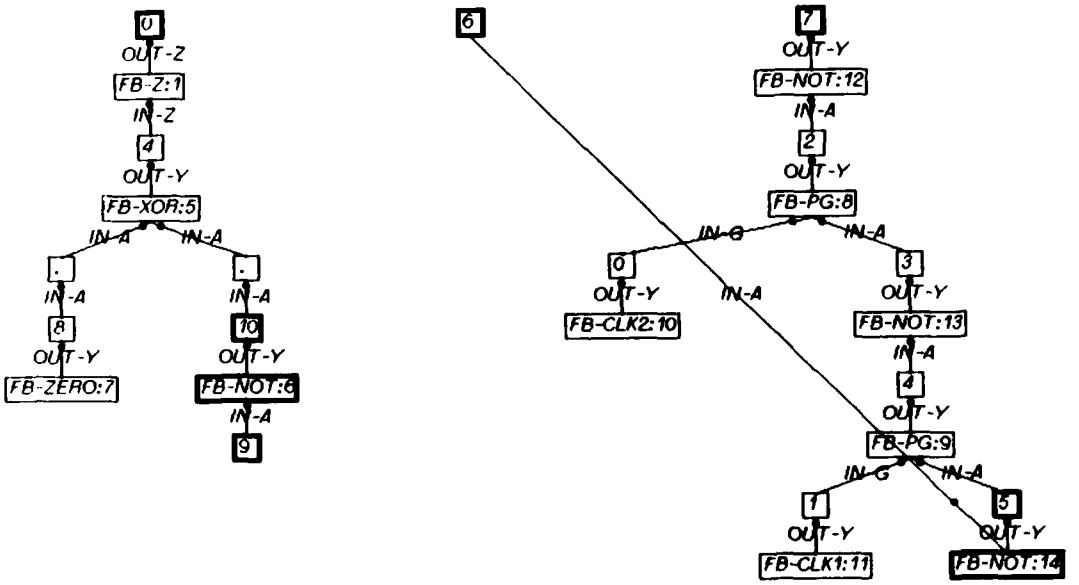


Figure E.39: After Eight Steps

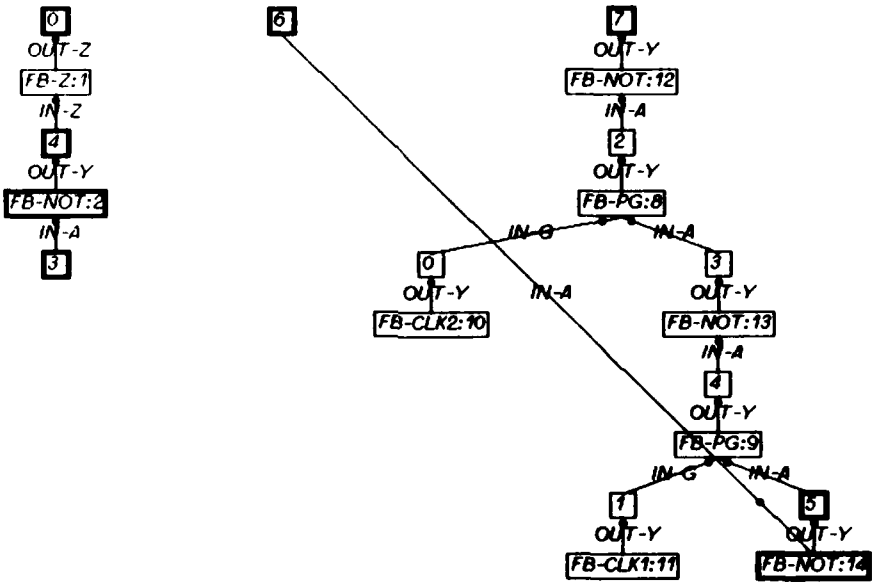


Figure E.40: After Nine Steps

E.3 Summary

- The depth-first approach taken in the original system is severely flawed, in that it often causes the system to find unnecessarily long and inefficient derivations. This leads to a large increase in run-time. Breadth-first search alleviates this problem.
- The *SEARCH-DEPTH* parameter allows a tradeoff between length of time searching and power of the algorithm. With this parameter set too small, the system will fail to explain some precedents which it could otherwise explain.
- The grammar rules available to the system have an enormous effect on the speed of the system. Having too many useless grammar rules is bad, and an inability to focus attention on a small portion of the problem is also.
- The system can be too greedy, by accepting the first step it sees which makes progress. This can cause it to miss a valuable rule application.

Bibliography

[1] Peter M. Andreae. Constraint limited generalization: acquiring procedures from examples. In *Proceedings of the 1984 AAAI Conference*, Amer. Assoc. for Artificial Intelligence, 1984.

[2] Robert C. Berwick. *The Acquisition of Syntactic Knowledge.* MIT Press, Cambridge Mass., 1985.

[3] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3, 1972.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., 1979.

[5] Robert Joseph Hall. *On Using Analogy to Learn Design Grammar Rules.* Master's thesis, Massachusetts Institute of Technology, 1985.

[6] R. M. Haralick and L. G. Shapiro. The consistent labeling problem: part i. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-1(2), April 1979.

[7] R. M. Haralick and L. G. Shapiro. The consistent labeling problem: part ii. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-2(3), May 1980.

[8] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[9] S. Mahadevan. Verification-based learning: a generalization strategy for inferring problem-reduction methods. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, IJCAI-85, 1985.

[10] Ryszard S. Michalski and Robert E. Stepp. *Machine Learning*, chapter 12. Tioga Publishing Company, 1983.

[11] T.M. Mitchell, P.E.Utgoff, and R.B.Banerji. Learning problem-solving heuristics by experimentation. In *Machine Learning*, Tioga Publishing, 1983.

[12] Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. *Explanation-Based Generalization: A Unifying View.* Technical Report ML-TR-2, SUNJ Rutgers, 1985.

[13] Raymond Mooney and Gerald DeJong. Learning schemata for natural language processing. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 1985.

[14] Shankar Rajamoney, Gerald DeJong, and Boi Faltings. Towards a model of conceptual knowledge acquisition through directed experimentation. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 1985.

[15] Andrew Lewis Ressler. *A Circuit Grammar for Operational Amplifier Design.* Technical Report TR-807, Massachusetts Institute Technology, 1984.

[16] Robert Sedgewick. *Algorithms.* Addison-Wesley, 1983.

[17] Reid G. Smith, Howard Winston, Tom M. Mitchell, and Bruce G. Buchanan. Representation and use of explicit justifications for knowledge base refinement. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 1985.

[18] Guy L. Steele. *Common Lisp: The Language.* Digital Press, 1984.

[19] Louis I. Steinberg and Tom M. Mitchell. A knowledge based approach to vlsi cad: the redesign system. In *Proceedings of the 21st Design Automation Conference*, IEEE, 1984.

[20] J. R. Ullman. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23(1), January 1976.

[21] Patrick H. Winston, Thomas O. Binford, Boris Katz, and Michael Lowry. *Learning Physical Descriptions from Functional Definitions, Examples, and Precedents.* Technical Report AIM-679, Massachusetts Institute of Technology, 1983.

[22] Patrick Henry Winston. *Learning by Understanding Analogies.* Technical Report AIM-520, M.I.T., 1979.

END

1-87

DTIC